A Pruning Algorithm for Mining Long and Maximal Length Frequent Itemsets

Sina Lessanibahri[a,b] , Camino González Fernández[b] , Luca Gastaldi[a]

[a] Politecnico di Milano(Polimi) — Department of Management, Economics and Industrial Engineering

[b] Universidad Politécnica de Madrid (UPM) —Department of Industrial Engineering, Business Administration and Statistics

Abstract. Association rule mining is one of the most popular exploratory data mining techniques to discover interesting and previously unknown correlations from datasets. However, current algorithms for association rule mining are computationally expensive, especially for very large datasets. In this paper, we introduce a novel algorithm, called LengthSort that reflects over the additional hidden information in the transactions' length. This algorithm is especially designed for mining long and maximum length frequent itemsets and it reduces the search space by pruning both items and transactions before constructing the Frequent Pattern tree.

Key words: Association Rules Mining; Maximum length frequent itemsets; data mining; long frequent itemsets

1 Introduction

Helped by the recent advances in technology, nowadays many businesses produce and store massive amounts of data. The rise in these big datasets highlights the importance of using intelligent techniques to analyze data effectively. Association rule mining [1] is one of the most popular exploratory data mining techniques for extrapolating interesting and previously unknown patterns from large volumes of data. Despite having originally been developed for market basket analysis purposes [2], recently there has been growing attention towards using these association rules in many other applications, such as in medical datasets [3]. These association rules are used principally to find the most frequent values of a set of variables (discovery of frequent itemsets), and then detect the relationships between the frequent items (rule generation) [4]. Of the two, the discovery of frequent itemsets is usually more expensive in terms of computational requirements [4].

Since the introduction of the Apriori algorithm [5], association rule mining has been improving significantly, over the past two decades. Among others, Han et al. [6], introduced an algorithm that discovers the association rules without candidate generation step required by Apriori. Deng et al.

[7] introduced the concept of N-List that represents the transaction datasets more efficiently and decreases the time required for the mining process. However association rule mining are not, as yet, applicable to very large datasets [8,9] or else they can be applied only to larger values of frequency thresholds. Unfortunately, in many real-world applications, the rules generated with lower frequencies have more interesting implications [e.g., 6]. For instance, rules with lower support are useful in detecting anomalies with applications in security systems and network intrusion detection [11] or discovery of patterns pertaining to purchases of expensive or luxury products. Thus, further improvements are needed to mine large datasets in view of finding the interesting associations more efficiently.

Moreover, datasets that include a large number of items usually produce many frequent itemsets. The discovered frequent itemsets thus need to be further analyzed by experts to detect those that are interesting. Many of these itemsets, especially the ones with smaller lengths, are less interesting in practice [4] as they are redundant [12] and are included in the more representative itemsets. To overcome this problem, researchers have introduced the concepts of Maximal Frequent Itemset (MFI), which are frequent itemsets that have no frequent supersets [8, 9] and Frequent Closed Itemsets (FCI), which are frequent itemsets that have no superset with the same support level [15,16]. In a similar vein, Hu et. al. [17] worked on Maximum Length Frequent Itemsets (LFI). LFI denotes the set of all frequent itemsets containing the maximum number of frequent items. LFI can be used in a number of applications, including to provide tours in the tourism field, in insurance coverage packages and in association rule-based clustering [17]. A further potential application is in healthcare. LFI can be used to detect sets of patients with the most similar conditions and/or health status. Considering patients' characteristics and medical conditions as items, through discovering the LFI, we are able to detect the set of the patients that are most similar. For instance, the analysis of these set of patients has interesting implications in terms of evaluating the effectiveness of various treatments. Similarly, comparing the services delivered and resources allocated to the set of similar patients in different geographical areas or hospitals, reveals inefficiencies and the so-called unwarranted variations.

The main contributions of this study are: (1) We proposed a novel algorithm for mining LFI. Our experiments on variety of datasets, demonstrate that the introduced algorithm is in most cases faster that the state of the art algorithms for mining LFI. (2) we introduced set of novel pruning

techniques and the corresponding algorithm named LengthSort, that reduces the size of the dataset prior to the frequent itemset mining. The reduced dataset can be the input of other mining algorithms to discover maximum length or long frequent itemsets. Thus, LengthSort can be coupled with many other state of the art algorithms to improve the efficiency when the aim is discovering the long frequent itemsets.

The remainder of the paper is organized as follows. In Section 2, we have briefly discussed the frequent itemsets mining problem statement and reviewed the related algorithms. In Section 3, we have explained LengthSort, the algorithm we are proposing. In Section 4, we have highlighted the results of a comparison between LengthSort and the two state-of-the-art algorithms used in LFI mining, LFIMiner and MaxLFI. As LFIs can be also discovered by mining MFIs, we also compared our results with the state-of-the-art MFI mining algorithm, namely INLA-MFP [12]. Finally, in Section 5, we have discussed the implications of the study, the limitations of the algorithm and several fruitful directions for further research.

2 Preliminaries and related work

In this section, we have recapped the problem statement and the two main search strategies for mining frequent itemsets. Then, we have briefly reviewed the state-of-the-art regarding algorithms for LFI mining.

2.1 Preliminaries

The problem of finding the most frequent variables was first introduced in [1] to analyse market basket data and where the variables can only assume binary values. The variable assumes the value 1 if the item is present in the corresponding record (transaction) and is set at 0 otherwise. For instance, consider a dataset with N_p or so variables called items, $X = (X_1 \dots X_{N_p})$. In this setting, the goal is to find a subset of variables $I \subseteq \{1 \dots N_p\}$ called ***itemset*** such that the probability given in Eq. 1 is greater than a given threshold [18]:

$$Supp(I) = Pr\left[\bigcap_{j \in I}(X_j = 1)\right] = \frac{1}{N_T}\sum_{k=1}^{N_T}\prod_{j \in I} X_{kj} \quad (1)$$

where N_T is the total number of data records (transactions) in the dataset and X_{kj} represents the value of variable *j* in record *k*. The probability defined in Eq.1 is known as the ***support*** of itemset *I* and is denoted as $Supp(I)$. In other words, the support of an item is its frequency divided by the total number of transactions. We refer to the frequency of an item or itemset as its *support count* or simply *count*. An itemset whose support is greater or equal to a user specified minimum threshold is called a ***frequent itemset***. Agrawal et. al. [5] developed an algorithm, called the ***Apriori*** algorithm, to find frequent sets in a dataset. This algorithm exploits what is known as the ***Apriori Principle***, which states that, any subset of a frequent itemset is also a frequent itemset [5].

The ***Apriori*** algorithm follows an approach known as the "level-wise search" for finding the frequent itemsets. In this approach, itemsets of size K are used to discover frequent item sets of size (K+1). Based on the Apriori principle, if a (K+1) itemset is frequent, then all its subsets of size K are also frequent. Thus, a (K+1) itemset is a candidate itemset only if there are two frequent subsets of size K whose first (K-1) items are in common. Thus, the *Apriori* algorithm starts by finding all the frequent sets of size 1, then the 1-itemsets are used to generate candidate 2-itemsets and so on.

Apriori and Apriori-based algorithms are breadth-first approaches that apply a "bottom-up" search logic to mine the frequent itemsets. The drawback to these types of algorithms is the fact that they produce candidate itemsets and all these candidates are checked to detect the frequent itemsets. This requires several scans of the dataset, especially for longer frequent itemsets.

Han et al. [6] introduced a **Frequent Pattern-growth (FP-growth)** algorithm which avoids the issue of generating candidates and only scans the dataset twice. The first scan detects the single frequent items and sorts them into descending order, according to their support level. The ordered items are stored in a ***header table*** (H-table). The FP-growth algorithm and its variants are then used to mine the frequent-pattern tree (**FP-tree)**, which is a compact tree structure of the dataset that is built during the second scanning process. In addition to the support, the header table stores a ***node-link*** for each item. For any given item, the node-link can be used to identify all the corresponding nodes in the tree. The FP-tree and H-table store all the necessary information to mine the frequent itemsets without generating candidate itemsets. Figure 1 shows the example of a dataset and its corresponding FP-tree and H-table.

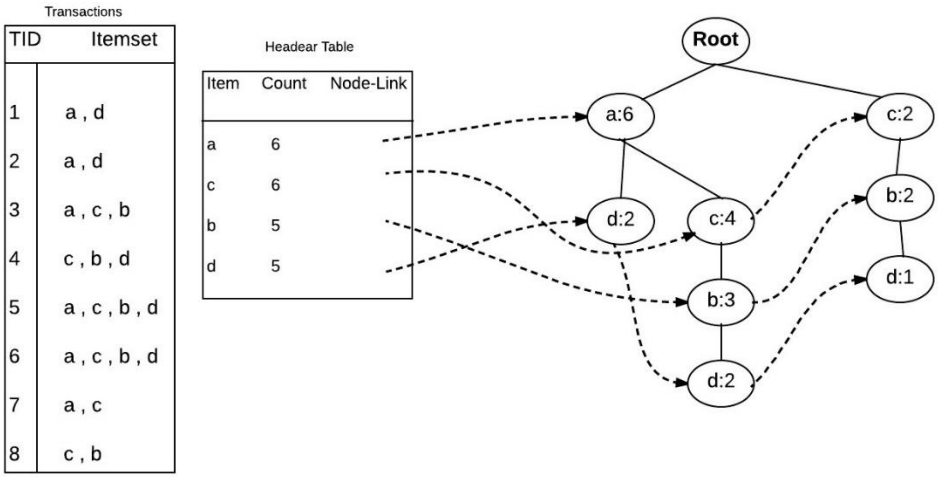


Figure 1: Example of an FP-tree structure

The set of transactions that contain the itemset *I* is called the ***conditional pattern base*** of *I*. For each item in the H-table, the FP-growth algorithm extracts the conditional pattern and constructs a ***conditional FP-tree***. When building the conditional FP-tree, the algorithm continues recursively to mine all the frequent itemsets. As shown in previous studies, the FP-growth and its variants are computationally more efficient than in Apriori-like algorithms, especially for data sets containing longer frequent itemsets. However, as the algorithm works recursively, the FP-tree needs to be fitted into the main memory of the computer being used.

Deng et al.[7] introduced PrePost algorithm that uses PPC-tree instead of FP-tree. In PPC-tree instead of node-links, items are associated with pre-order and post-order fields that are assigned based on the tree traversals. In pre-order traversal, the ranks of the parent nodes are smaller while, the ranks of the children are smaller in post-order traversal. The pre and post order traversals start from left branches to the right. Using pre and post-orders, H-table is not needed to mine the frequent itemsets. After construction of PPC-tree, each item is associated with a sequence of tuples, called N-List. N-List of an item consists of pre-order, post-order and count of all the tree nodes corresponding to the item. PrePost then uses the N-Lists to discover frequent itemsets in a level-wise manner. N-List structure is more compact than FP-tree and can improve the runtime of frequent itemsets mining.

2.2 Related work

To the best of our knowledge, LFIMiner [17] and MaxLFI [19] are currently the most efficient algorithms for mining the LFI. LFIMiner is an extension of the FP-growth algorithm, and it applies three techniques to reduce the search space. After constructing the initial FP-tree, LFIMiner employs a procedure known as *Conditional Pattern-base Pruning (CPP)*. While constructing the conditional pattern base of the items, any conditional transaction whose length is smaller than the length of the longest frequent itemset found so far is discarded and will not be included in the subsequent conditional FP-tree. If the surviving conditional transaction is also frequent and its length is longer than the current length, then the pruning length is updated.

In the next step, the algorithm incorporates a further process known as *Frequent Item Pruning (FIP)*. FIP acts in a similar way to CPP, pruning the conditional transactions based on the length of the longest frequent itemset found up to that point. However, FIP removes all conditional transactions that do not contain enough frequent items. More specifically, FIP finds all the items that are frequent in the conditional pattern base, and then the conditional transactions with fewer frequent items than the current pruning length are trimmed. FIP is applied recursively, since, by eliminating the conditional transactions, other items may become infrequent, which, in turn, can result in other transactions not being long enough.

Lastly, LFIMiner applies ***dynamic reordering*** [20] to the items in the header table, so that the conditional trees becomes more compact. In other words, the header tables of all the conditional trees are sorted according to the support of the items in the corresponding conditional pattern base.

As mentioned in Section 2.1, while constructing the FP-tree, items are sorted in support-descending order in the header table, as well as in the set of transactions. Consequently, the same order is preserved within the conditional patterns and paths in the FP-tree. MaxLFI [19] is an extension of LFIMiner that exploits the order of the items to improve its pruning techniques. The first improvement is based on the observation that the conditional pattern base of an item in the i_{th} position (top-down order) of the header table cannot contribute to frequent itemsets which are longer than i. The authors in [19] also introduced the *Max Position* of the items, defined as the maximum position of a given item in the set of sorted transactions. The *Max Position* of the items is recorded in an additional column in the header table during the construction of the initial FP-

tree. The conditional transactions for items with Max Position *t* cannot generate frequent itemsets longer than *t.* These two techniques are the bases of what is known as Pre-pruning CPP (PreCPP) in MaxLFI algorithm.

The effectiveness of the above mentioned pruning techniques (CPP, FIP and PreCPP) relies on the length of the longest frequent found so far. Thus, after constructing the initial FP-tree, MaxLFI scans the nodes of the tree (through the header table and node links) whose count is greater than the support threshold, in order to gain an estimate of the longest frequent itemset, prior to the pruning and the construction of the conditional FP-trees. For more detailed information on LFIMiner and MaxLFI, refer to [17] and [19].

LFI mining can be also performed by applying MFI algorithms. MFI are in fact subset of LFI. So discovering MFI and then pruning the shorter frequent itemsets afterwards, is an alternative approach to discover LFI. To the best of our knowledge, INLA-MFP [12] is the state-of-the-art algorithm for extracting MFI. INLA-MFP employs N-List structure and children-parent equivalence [21] pruning, to discover the MFI more efficiently.

MaxLFI, LFIMiner and INLA-MFP start the processes of pruning and estimating the maximum length, after having constructed the initial tree. However, building a tree that includes all the transactions and frequent items is computationally expensive, especially for large datasets. Moreover, in MaxLFI and LFIMiner the computational effort of constructing the subsequent conditional trees will increase as the size of the initial tree grows. Similarly, as the size of the initial tree increases, the number and cardinality of the N-Lists rises and consequently the cost of performing intersections needed in INLA-MFP increases.

In this paper, we have proposed an approach for pruning both transactions and items before building the initial tree and consequently improve the efficiency of the LFI mining. Lastly, since the tree needs to be fitted into the main memory of the user's computer, reducing the size of the initial tree is crucial to improve the scalability of the procedure up to very large datasets.

In the rest of the study, we refer to the number of items present in the transaction, as the size or ***length of the transaction***. Similarly, the length of an itemset is defined as the number of the items contained in the given itemset. Thus, the length of an itemset or transaction is the cardinality of the set or the sum of the corresponding record in the binary representation.

3 Mining LFI with LengthSort

In this section, we have introduced the methods that we use to estimate the maximum length of the frequent itemsets and prune the dataset items and transactions prior to constructing the FP-tree accordingly. We have also presented a modified FP-tree structure that can be updated with new transactions and items in the event that the estimated maximum length is not equal to the actual length.

3.1 Pruning the dataset before constructing the tree

The LengthSort algorithm exploits information embedded in the variations within the transactions' lengths, which exists for the very fact that there are these variations. The process starts by sorting the transactions according to their lengths in descending order. Then, based on these lengths, the dataset is partitioned, so that each partition consists of transactions of the same length. The ***cumulative support*** of each variable is calculated from the partitions, going from the longest length to the shortest. The cumulative support of a given item in the partition of length K is the support of the item in that partition plus the support of the item in the partitions with longer lengths. Example 1 illustrates the partitioning process.

Example 1. As shown in Figure 2.a, the original dataset contains 11 items and 14 transactions. The last column shows the transaction lengths. The transactions are sorted according to their lengths, and same-length rows are used to construct the partitions, as illustrated in Figure 2.b. Finally, the cumulative counts of the items are calculated for each partition. Note that the cumulative count (support) of an item in the last partition (partition 7 in Figure 2. c) is the total count (support) of the item in the dataset. ■

Trans. ID	Item1	Item2	Item3	Item4	Item5	Item6	Item7	Item8	Item9	Item10	Item11	Trans. Length
1	1	1	0	0	1	1	1	0	1	1	0	**7**
2	1	1	1	1	0	1	1	0	0	0	1	**7**
3	1	1	1	1	0	0	1	0	0	0	1	**6**
4	1	1	1	1	0	0	0	0	1	0	0	**5**
5	0	1	1	1	1	0	1	0	0	0	1	**6**

6	1	1	1	1	1	0	1	1	0	0	1	**8**
7	1	1	1	1	1	0	0	1	0	0	0	**6**
8	1	1	0	1	0	0	0	0	0	0	0	**3**
9	0	0	1	1	0	0	0	0	0	0	0	**2**
10	0	1	0	0	0	1	0	0	0	0	0	**2**
11	1	0	0	0	0	0	1	0	0	0	0	**2**
12	0	0	1	1	0	0	0	0	0	0	0	**2**
13	1	0	0	0	0	0	0	0	0	0	0	**1**
14	0	1	0	0	0	0	0	0	0	0	0	**1**

a) Dataset of Example 1

Trans. ID	Item1	Item2	Item3	Item4	Item5	Item6	Item7	Item8	Item9	Item10	Item11	Trans. Length	Partition
6	1	1	1	1	1	0	1	1	0	0	1	8	**1**
1	1	1	0	0	1	1	1	0	1	1	0	7	**2**
2	1	1	1	1	0	1	1	0	0	0	1	7	
3	1	1	1	1	0	0	1	0	0	0	1	6	**3**
5	0	1	1	1	1	0	1	0	0	0	1	6	
7	1	1	1	1	1	0	0	1	0	0	0	6	
4	1	1	1	1	0	0	0	0	1	0	0	5	**4**
8	1	1	0	1	0	0	0	0	0	0	0	3	**5**
9	0	0	1	1	0	0	0	0	0	0	0	2	**6**
10	0	1	0	0	0	1	0	0	0	0	0	2	
11	1	0	0	0	0	0	1	0	0	0	0	2	
12	0	0	1	1	0	0	0	0	0	0	0	2	
13	1	0	0	0	0	0	0	0	0	0	0	1	**7**
14	0	1	0	0	0	0	0	0	0	0	0	1	

b) Sorting transactions based on length and the partitioning of the dataset

	Item1	Item2	Item3	Item4	Item5	Item6	Item7	Item8	Item9	Item10	Item11	Partition Length
Partition 1	1	1	1	1	1	0	1	1	0	0	1	**8**
Partition 2	3	3	2	2	2	2	3	1	1	1	2	**7**
Partition 3	5	6	5	5	4	2	5	2	1	1	4	**6**
Partition 4	6	7	6	6	4	2	5	2	2	1	4	**5**
Partition 5	7	8	6	7	4	2	5	2	2	1	4	**3**
Partition 6	8	9	8	9	4	3	6	2	2	1	4	**2**
Partition 7	9	10	8	9	4	3	6	2	2	1	4	**1**

c) Cumulative counts for each item in the partitions

Figure 2. The process of sorting and partitioning

In order to prune the dataset and discover the LFI more efficiently, we estimated an initial upper bound for the maximum length of the frequent itemsets. The initial upper bound estimation is obtained through the following Lemma:

Lemma 1 (first upper bound): Assume a dataset partitioned as previously explained, with a maximum partition size P. The possible P partition lengths are denoted by K_i ; $i \in \{1 \dots P\}$. The number of items in partition K_i with a cumulative support that is greater than the minimum support

is denoted by F_i. K^* is defined as $Max\ \{K_i | K_i \leq F_i\}$ and the maximum length is then less than or equal to K^*.

Proof: By contradiction, we assume that L (the maximum length) is greater than K^* Thus, the support of the set of L items is greater than or equal to the minimum support. It is trivial that the itemset is also frequent in a set of transactions whose length is equal to or greater than L. Furthermore, according to the Apriori principle, the individual supports of these L items are also greater than the minimum support. Thus, we have a partition size L ($1 \leq L \leq$ P) which has L frequent items. But $L > K^*$, thus K^* cannot be the $Max\ \{K_i | K_i \leq F_i\}$, which contradicts the assumption. ■

Example 2. Consider the dataset in Example 1 and a minimum support count of 4. The number of frequent items in each partition is shown in Figure 3.

	Length		No. Frequent Items
Partition 1	8	≰	0
Partition 2	7	≰	3
Partition 3	6	≤	7
Partition 4	5	≤	7
Partition 5	3	≤	7
Partition 6	2	≤	7
Partition 7	1	≤	7

Figure 3. The initial upper bound estimated through Lemma 1 for the dataset in Example 1

The length is 6 for partition 3, and it is the largest length for which the number of frequent items is greater than the length. Thus, in this example, the initial upper bound for the maximum length is 6. It important to note that the inequality of Lemma 1 needs to be checked for every potential partition size, not just for the partitions that were actually found in the dataset. For instance, in Figure 3, there is a jump from partition length 5 to 3. Thus, we need to check whether $K_4 = 4$ is equal to or smaller than $F_4 = F_5 = 7$. To note, it obviously does not matter in this example, as the

upper bound was already found to be 6 and there is no need to check for any potential (or actual) partitions whose lengths are less than 6. ■

Thus, using Lemma 1, it is possible to calculate the initial upper bound for the maximum lengths. Figure 4 illustrates the algorithm for estimating the first upper bound.

```
Algorithm: Filter 1 (First upper bound)
Input: dataset, support threshold
Output: dataset with eliminated rows and Items, current length

Delete items with support < support threshold
Sort and partition the transactions by their lengths
For each partition(i) in the dataset (from top to bottom)
        Calculate the cumulative support of the items in the partition(i)
        Calculate Np = number of items with cumulative support >= threshold
        If Np >= Partition(i).length
                current length = partition(i).length
                Return current length, sorted dataset
        End if
        Jump=partition(i).length - partition(i+1).length
        If Jump >1
                For each j from 1 to (Jump-1)
                        If Np >= partition(i).length - j
                                current length = partition(i).length-j
                                Return current Length, sorted dataset
                        End if
                End for
        End if
End For
```

Figure 4: Pseudo code of Filter1 (First upper bound) Algorithm

The initial upper bound obtained through Lemma 1 may not be close enough to the actual maximum length for the larger real-world datasets. The initial upper bound can be improved further by using Lemma 2.

Lemma 2 (Second upper bound): If there is a frequent itemset of length K_i, then F_i remains greater than or equal to K_i after eliminating all the non-frequent (according to the cumulative support) items in the partition of size K_i

Proof: If there is a frequent itemset of size K_i, through the Apriori principle, we know that all the items in that frequent itemset are also frequent and since, only the non-frequent items have been eliminated, we have $F_i \geq K_i$.

■

So, by eliminating the non-frequent items in the partition corresponding to the size of initial upper bound, we can check whether the initial upper bound holds. This is especially effective since it is performed iteratively. First, all the non-frequent items in the partition that are the same size or greater of the first upper bound are eliminated from the dataset. If some of the transactions fall into smaller partitions and more items become infrequent, then the process is repeated. If the initial upper bound decreases after the iterations ($F_i \ngeq K_i$) , this means there are no frequent itemsets of that size and we can move to the next smaller length (K_i-1) and repeat the process using the updated length. As the length is reduced, some of the previously pruned items and/or transactions will be restored.

Example 3 illustrates the application of Lemma 2.

Example 3. Consider the dataset in Example 1**.** By using Lemma 2, all the non-frequent items in partitions of length 6 can be eliminated. The process of sorting by length and partitioning is performed once again. The results are illustrated in Figure 5.

	Item1	Item2	Item3	Item4	Item5	Item6	Item7	Item8	Item9	Item10	Item11	Length
Partition 1	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**0**	**0**	**1**	**8**
Partition 2	**3**	**3**	**2**	**2**	**2**	**2**	**3**	**1**	**1**	**1**	**2**	**7**
Partition 3	**5**	**6**	**5**	**5**	**4**	**2**	**5**	**2**	**1**	**1**	**4**	**6**
Partition 4	**6**	**7**	**6**	**6**	**4**	**2**	**5**	**2**	**2**	**1**	**4**	**5**

Partition 5	**7**	**8**	**6**	**7**	**4**	**2**	**5**	**2**	**2**	**1**	**4**	**3**
Partition 6	**8**	**9**	**8**	**9**	**4**	**3**	**6**	**2**	**2**	**1**	**4**	**2**
Partition 7	**9**	**10**	**8**	**9**	**4**	**3**	**6**	**2**	**2**	**1**	**4**	**1**

a) Partitions and the cumulative supports of the items. Items 6, 8, 9 and 10 have a cumulative count of less than 4 in partition 3 and can be omitted by means of Lemma 2.

	Length	No. Frequent Items
Partition 1	7 $\not\leq$	0
Partition 2	6 $\not\leq$	5
Partition 3	5 $\leq$	6
Partition 4	4 $\leq$	7
Partition 5	3 $\leq$	7
Partition 6	2 $\leq$	7
Partition 7	1 $\leq$	7

b) After discarding the infrequent items, the dataset is sorted and partitioned and the new upper-bound is estimated

Figure 5. Process of improving the initial upper bound by means of Lemma 2

After removing the infrequent items, the upper bound has decreased to 5. Based on Lemma 2, there is no frequent itemset of size 6 and the new upper bound is 5. ■

Figure 6 depicts the algorithm for estimating the second upper bound.

```
Algorithm: Filter 2
Input: dataset, support threshold, current length
Output: pruned dataset, updated current length
Let pruned dataset ←dataset
While any transaction or item is deleted from pruned dataset
    Delete items with support < threshold
    Delete transactions with length < current length
    If number of items < current length
        current length = current length - 1
        Return current Length, pruned dataset
    End if
End While
Return current length , pruned dataset
```

Figure 6: Pseudo code of Filter 2 (Second upper bound) Algorithm

As illustrated in the Figure 6, we have pruned both the items and the transactions from the dataset. It should be noted that, with the help of the estimated maximum length, it is also possible to prune the dataset vertically. As also highlighted in [22], the transactions whose length is less than the estimated maximum length are not able to contribute to LFI, so they can be omitted from the dataset.

Use of the first two Lemmas was sufficient to derive an exact estimate of the upper bound in the example but, for the larger datasets with numerous items, a further improvement to the upper bound is needed. Furthermore, the number of candidate items that potentially can form an element in LFI should be further pruned to mine the maximum frequent itemsets efficiently. Lemma 3 further improves the upper bound.

Lemma 3: If there is a frequent itemset of size K_i and v is an item in that frequent set, after removing v from the dataset, for at least $(K_i - 1)$ items, the sum of the cumulative supports of the partitions with lengths equal or greater than K_i is reduced by an amount equal or greater than the minimum support threshold. In other words, the following inequality

$$\sum_{l\in\{m|P_m\geq K_i\}}\left(CSupp_{jl}^{B} - CSupp_{jl}^{A}\right) \geq Min\ Supp \qquad j \in \{items\} \qquad (2)$$

holds for at least $K_i - 1$ items where:

- $CSupp_{jl}^{B}$ *is the cumulative support of item j in partition l* ***before*** *eliminationg item v,*
- $CSupp_{jl}^{A}$ *is the cumulative support of item j in partition l* ***after*** *eliminationg item v,*
- $\{m|P_m \geq K_i\}$ *is the set of indices of partitions with length equal or greater than* K_i *before eliminating item v.*
- $Min\ Supp$ *is the minimum support treshold*
- $Min\ Count$ *is the minimum count treshold*

Proof: Since item v is in a frequent itemset with length equal to K_i, there are at least $Min\ Count$ transactions that simultaneously contain *v* and $(K_i - 1)$ other items. By removing item *v,* the length of each of these transactions is reduced by one and, consequently, the transactions are moved to the next partition with a shorter length. It is trivial that these are transactions with lengths equal to or greater than K_i.Thus, for each of the $(K_i - 1)$ items, at least $Min\ Count$ transactions are transferred to the lower partitions. Hence, after removing v , for each item in the frequent itemset, the sum of cumulative supports in the partitions that are greater than or equal to K_i plus $Min\ Count$

is smaller or equal to the sum of the cumulative supports of the same item before the elimination process:

$$\sum_{l\in\{m|P_m\geq K_i\}}\left(CCount_{jl}^{A}\right)+Min\ Supp\leq\sum_{l\in\{m|P_m\geq K_i\}}\left(CCount_{jl}^{B}\right)\qquad for\ K_i-1\ items$$

$$\Rightarrow\sum_{l\in\{m|P_m\geq K_i\}}\left(CCount_{jl}^{B}-CCount_{jl}^{A}\right)\geq Min\ Count\qquad for\ K_i-1\ items$$

Dividing the inequality by N_T, the total number of transactions, we obtain Eq. 2 ■

By using Lemma 3, we can eliminate more of the items that are not in a frequent itemset of a given length. This is achieved by eliminating the items one by one and calculating the sum of the cumulating support of the other items in partitions with lengths greater or equal to the given length. This process is performed on smaller subset of the dataset. Example 4 shows the application of Lemma 3 to the dataset shown in the example.

Example 4. Following the previous example, the upper bound is 5 and there are six frequent items in the partition with length 5. So, if there is any frequent set of size 5, it is also a subset of the set of these six items. Instead of checking all the subsets, we can use Lemma 3 and further prune the candidate items. The six individually frequent items are eliminated one by one from the data and the differences between the cumulative supports of other items are then calculated.

		Difference between the sums of cumulative counts						Number of items whose count is reduced by more than the minimum support
		Item 1	Item 2	Item 3	Item 4	Item 7	Item 11	
Eliminated Items	Item 1		4	4	4	3	3	**3**
	Item 2	4		5	5	4	4	**5**
	Item 3	4	5		5	4	4	**5**
	Item 4	4	5	5		4	4	**5**
	Item 7	3	4	4	4		4	**4**
	Item 11	3	4	4	4	4		**4**

Figure 7. Pruning the candidate items based on Lemma 3

The counts in each row of Figure 7 are calculated from Eq. 2. The last column shows the number of items for which the sum of cumulative counts has been reduced by more than the minimum count (4). As can be seen in the first row, after eliminating Item 1, the sum of the cumulative counts for three items was reduced by an amount that is greater than the minimum threshold. Thus, according to Lemma 3, Item 1 cannot be in a frequent itemset of size 5. This means that the set {Item 2, Item 3, Item 4, Item 7, Item 11} is the only remaining candidate set of length 5. By scanning the dataset, this set is confirmed as the only frequent itemset of size 5 and, hence, the only element in the LFI. ■

Lemma 3 is especially useful in combination with Lemma 2. After items have been eliminated through Lemma 3, we can use the Lemma 2 algorithm to further prune both the items and the transactions. Figure 8 shows the algorithm for pruning the items using Lemma 3. It is straightforward to show that, for a given item j, after removing the item v , the value of the left-hand side of Eq. 2 is in fact the v-conditional support in transactions with length greater than or equal to the partition size l. Thus, as illustrated in Figure 8, instead of calculating the left-hand side of Eq. 2 directly, we can calculate the conditional supports in the dataset where all the transactions that were not long enough have already been pruned using the Filter 2 algorithm.

```
Algorithm: Filter 3
Input: dataset, support threshold, current length
Output: updated current length, pruned dataset
Let pruned dataset ←dataset
For each Item i in pruned dataset
        Calculate the i-conditional support of all other items
        Calculate Ni:= Number of items with i-conditional support >= threshold
End for
Delete Items with Ni < current length - 1
If number of remained items < current length
        current length = current length - 1
End if
Return current length, pruned dataset
```

Figure 8: Pseudo code for the Filter 3 algorithm

As seen in the previous example, the estimated upper bound was equal to the maximum length. However, this is not a given for larger real-world datasets. In other words, some items that are not present in LFI, can still survive the pruning techniques of Lemmas 2 and 3. This results in estimating the maximum length that is greater than the actual length. For example, assume an item that is present in 5 transactions corresponding to a maximum length itemset and 6 transactions corresponding to another maximum length itemset. If the minimum count threshold is 11, the given item is not in any frequent itemset, but it will pass the pruning techniques of Lemmas 2 and 3. In this case, some of the removed items and/or transactions need to be restored and, therefore, we have suggested a modified FP-tree structure that allows this to be achieved more efficiently.

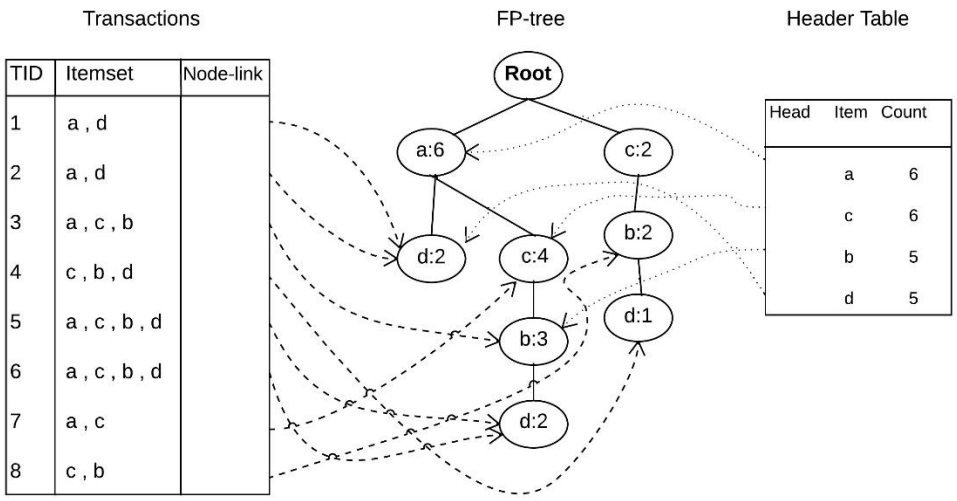
3.2 Constructing and updating the FP-tree

After obtaining the final estimate of the upper bound (when no further improvements to the estimate or reduction of the dataset are possible), we built the FP-tree and mined the reduced data set using the LFIMiner or MaxLFI algorithms. If the maximum length found is equal to the upper bound, then the upper bound found is accurate and no further mining is required. If the maximum length found is less than the upper bound, then we need to repeat the data pruning process with the updated length (the length of the longest frequent itemset found in the first run). The length found in the first run is, in fact, the lower bound, so, in the worst case scenario, we need to perform two runs of the algorithm to ensure that the results are complete. In order to improve efficiency in the second run, we have proposed a modified FP-tree structure, which can be updated with new transactions and items. So, instead of rebuilding the whole tree, we will simply add the pruned transactions and items that would not be eliminated using the new length.

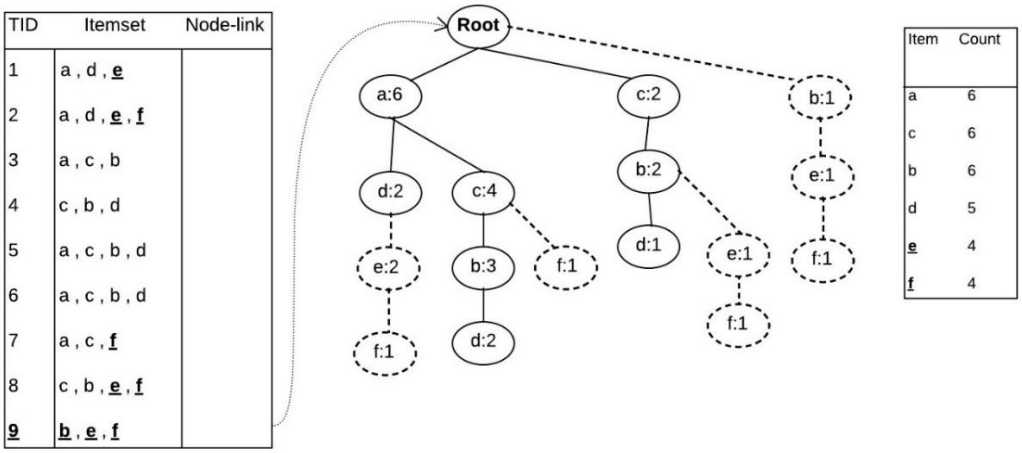
After repeating the pruning steps (Lemma 2 and 3) with the updated length and comparing the new reduced data set with previous one, the algorithm detects the transactions and items that need to be added to the initial FP-tree. Adding the new transactions is straightforward; we simply add them starting from the root node in the tree. In order to add new items, we considered a FP-tree structure with the transaction node-links. Therefore, in addition to the item node-links in the traditional FP-tree, we added a column to the dataset in which to store the transaction node-links. These node-links store the node in the tree that corresponds to the last item in the transaction. We

can add the new variables belonging to the transaction, starting from the node associated to the transaction's node-link.

In other words, in the event that the actual maximum length is less than the upper bound, the transactions eliminated in the first run whose lengths are greater than or equal to the new maximum length are added to the data. The new items (those that are not in the first tree header table) are then sorted according to their support and added to the end of the header table. The new transactions are inserted in the FP-tree, starting from the null node (Node-Link = 1), while, for the transactions with Node-Link ≠ 1, only new items are inserted in the tree, starting from each transaction's node-link. Figure 9 gives an example of updating an FP-tree with new transactions and items.



a) An FP-tree with transaction Node-link



b) The updated FP-tree. Transaction **9** and Items **e** and **f** are added to the dataset. The transaction is added starting from the root node and new items are added according to the transactions' node-links

Figure 9: Procedure of updating an FP-tree with transaction nodes.

Figure 10 illustrates the algorithm of the modified FP-tree.

```
Algorithm: Modified FP-tree
Input: dataset, support threshold, Optional: FP-tree0 (an initial FP-tree),
       H-table0(The initial H-table)
Output: FP-tree, transactions with node-link

Let transactions with node-link← Dataset
If FP-tree0 is empty
        Let FP-tree ← Ø
        Let H-table ← Ø
        Build the FP-tree from the dataset
        Add each transactions' last item node-link as the transaction node-link

Else
        Let FP-tree ← FP-tree0
        Let H-table ← H-table0
        Scan the dataset for new items that are not in H-table
        Sort the new items and add them to the H-table
        For each transaction
                If the transaction node-link == 1
                        Add the transaction to FP-tree from base node
                        Add the last item's node-link as the transaction node-link
        Else
                        Add the new items in the transaction to FP-tree starting from node-link
                        Add the last item's node-link as the transaction node-link
                End If
        End for
End If
Return FP-tree, H-table , transactions with node-link
```

Figure 10: Pseudo code of the modified FP-tree

Mining the constructed FP-tree can be performed using any selected algorithm. This means that our proposed algorithm can be added to the many tree-based frequent itemset mining procedures. In our experimental evaluation, we considered MaxLFI and LFIMiner. Figure 11 shows the LengthSort+LFIMiner algorithm; this is our proposed pruning algorithm to complement and improve the efficiency of LFIMiner used for mining frequent itemsets with a maximum length.

```
Algorithm: LengthSort+LFIMiner
Input: dataset, support threshold, current length
Output: LFI (maximum length frequent Itemset)
Call Filter 1 (dataset, threshold)
For i = 1:2
        While any item removed or current length Changed
                While current length reduced
                        Call Filter2
                End while
                Call Filter3
        End
        Add column of ones to dataset as the transactions' node-link
        Call Modified FP-tree
        Call LFIMiner [11]
        If actual length found < current length
                current length = actual length
                Add transactions with length >= current length to the data set
                Add 1 as the transactions' node-link to the new transactions
        Else
                Return LFI
        End if
End for
```

Figure 11: Pseudo code of LengthSort+LFIMiner algorithm

LengthSort+MaxLFI is similar to the algorithm presented in Figure 11, and only LFIMiner is replaced by MaxLFI. Similarly for LengthSort+INLA-MFP, LFIMiner is replaced with INLA-MFP [12] and instead of FP tree, we construct PPC-tree [7] in algorithm presented in Figure 11. The modified tree approach and transactions node-link can be applied to PPC-trees as well. And

finally, for mining frequent itemsets longer than a user-specified threshold, the call of Filter 1 is replaced with the user input for the minimum length of the frequent itemsets.

It is worth pointing out that improvement in efficiency is achieved, as LengthSort pre-prunes the dataset such that LFIMiner or MaxLFI will be used on a much smaller subset of the dataset.

In addition to FP-tree construction and LFIMiner, LengthSort+LFIMiner calls three other procedures, namely Filter 1, 2 and 3. The main operations of Filter 1 are: support counting, calculations of transactions' lengths and sorting the vector of the lengths. Assuming that N is the number of transactions and ω is the average transaction length, these operations require $O(N\omega)$, $O(N\omega)$ and $O(N \log N)$ respectively. Filter 2 performs support counting and calculating the transactions' lengths sp needs $O(N\omega)$ time. Operations of Filter 2 are performed on smaller proportion of datasets, which have smaller number of transactions and items. So $O(N\omega)$ is the worst-case scenario complexity. Algorithm of Filter 2 is repeated each time that items or transactions are removed or the current length estimate is updated. As the length is updated only if variables or transactions are pruned, the number of iterations is bounded by $L+V$, where L is the number of removed transactions and V is the number of pruned items. Thus, the complexity of all calls for Filter 2 is $O((L+V)N\omega)$. It is worth mentioning that in LengthSort, items and transactions are not necessarily pruned one at the time and we expect that several items and/or transactions are removed in each iteration. This specially happens more often in transactions' removal as they are pruned based on the length and usually there are several transactions with the same length in the datasets.

If M is the number of variables, Filter 3 calculates $M-1$ i-conditional supports where M is the number of items. The support counting is needed for transactions that contain item i, and is limited by the maximum support of the items, denoted as Φ. So, in worst-case scenario, each call of Filter 3 requires $O(M\Phi\omega)$. So Filter 3 is more sensitive to the number of survived items. Filter 3 is repeated much less than Filter 2 and it is called only if no further items or transactions can be pruned by Filter 2.

It can be noted that as $L+V$ increases, more time is spent by LengthSort. However, the more pruned transactions and items, the less computational effort is needed for constructing the tree and the rest of the mining process. So LengthSort can improve the computational effort of the itemsets mining, if the time complexity requirement of the mining algorithm for the pruned transactions

and items is more than the requirements mentioned above. In the case that LengthSort does not prune any items or transactions, it will add $O(N \log N + N\omega + M\Phi\omega)$ time to the mining algorithm.

Modified FP-tree algorithms adds one column to the dataset to store the transactions Node-Link. As the column is added to reduced dataset, this does not increase the space requirement unless, space required by $L \times V$ is less than N. In our implementation of filter 3, a matrix of size M^2 is created to hold the information of the i-conditional supports. Again this can increase the memory requirement only if the space requirement of the matrix is more than the initial tree.

4 Experimental evaluation

In this section, we have presented the experimental environment and the datasets used. The results of our experiments are presented for MaxLFI, LFIMiner, as well as for the LengthSort+MaxLFI and LengthSort+LFIMiner algorithms. We also performed experiments on INLA-MFP and LengthSort+INLA-MFP.

4.1 Experimental environment

The tests were performed using an Intel core i7, 2.4 GHz laptop with 12 GB main memory running Linux Ubuntu 16.4. The LengthSort, LFIMiner, MaxLFI, PPC-tree and INLA-MFP algorithms were implemented in MATLAB 2016b. To construct the FP-tree, we modified the MATLAB implementation of the FP-growth algorithm available at [23].

To evaluate the performance of the proposed algorithm, we carried out experiments on one synthetic and nine real-world datasets used in many previous studies (e.g., [1,8,16,24–27]). These are among the most widely used datasets to compare the performance of the frequent itemsets mining algorithms. The included datasets have different distributions in terms of number of transactions and items. The tests are performed on various support thresholds to assure the objectivity of the experiments.

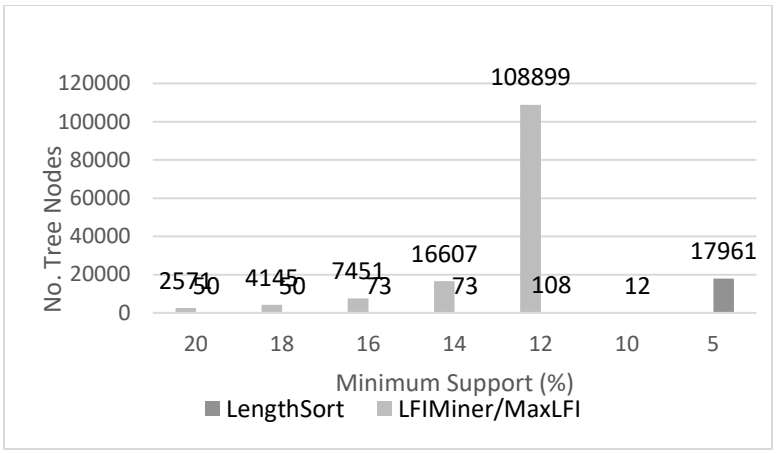
BMS-Web-View 1 and 2 include sequences of the click streams that were collected from a typical e-commerce webpage. The Groceries dataset presented in [28] contains one month's worth of real point-of-sale transactions. The dataset was gathered from a typical grocery store and contains 9,835 transactions. The purchased items are aggregated into 169 categories. Similarly,

the Retail [29] dataset contains anonymized transactions from a Belgian retail store. The Kosarak dataset contains sequences of click-streams from a news website. This dataset is large and consists of almost 1 million transactions. Chain Store is another large data set containing customer transactions in a given retail store [30]. T10I4D100K is a synthetic transaction dataset generated by the IBM Almaden Quest research group. Mushrooms, Chess and Susy are the datasets with more density. Susy is also the largest dataset considered, and includes five million transactions and 191 items representing the characteristics measured by particle detectors in the accelerator. The datasets used in the experiments, are publicly accessible from [31] and [32]. More details of the data sets are provided in Table 1.

Data set	No. Transactions	No. Items	Average Length	Standard Deviation of Length
Groceries	9,835	169	4.40	3.58
BMS-Web-View1 (Gazelle)	59,601	497	2.42	3.22
BMS-Web-View2	77,512	3,340	4.62	6.07
Retail	88,162	16,470	10.30	8.16
Kosarak	990,002	41,270	8.10	23.62
Chain Store	1,112,949	46,086	7.22	8.90
T10I4D100K	100,000	1,000	10.09	3.66
Mushrooms	8124	128	23	0
Chess	3196	76	37	0
Susy	5000000	191	18.58	0.49

Table 1: Datasets used to evaluate the performance of the LengthSort algorithm

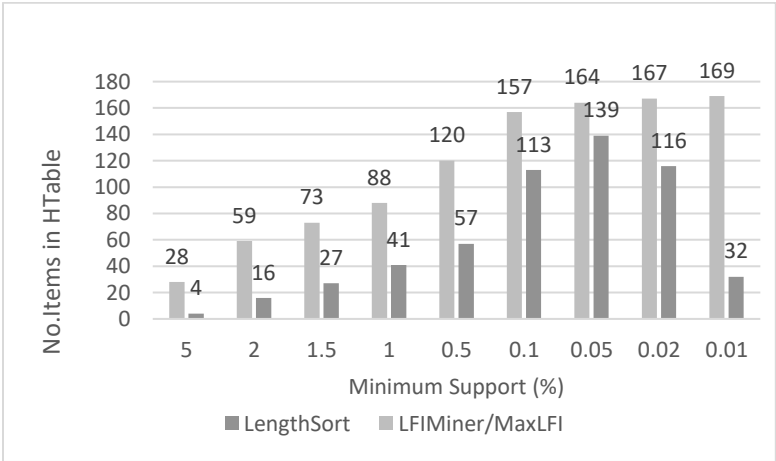
The datasets were encoded into sparse matrices for the LengthSort+LFIMiner and LengthSort+MaxLFI algorithms. The time needed for the encoding was added to the final runtime of these two algorithms. The output of the LengthSort was again transformed into transactional lists, which are the appropriate input for constructing the FP-tree. Even though this may not be a very efficient approach, we believe that it reveals how far LengthSort improves the performance, in spite of the time needed for dataset manipulation (i.e. if LengthSort is coupled with other

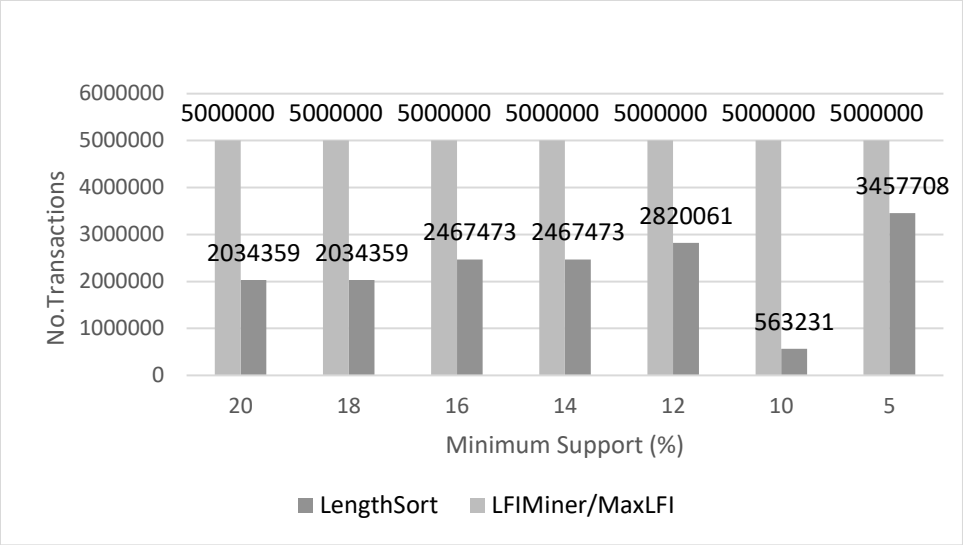


j) Susy

Figure 13: Number of initial tree nodes.

Figure 13 shows the initial FP-tree size in terms of number of tree nodes. The sizes of the initial FP-trees are identical for MaxLFI and LFIMiner, as they use no pruning heuristics prior to the construction of the initial tree. LengthSort+LFIMiner and LengthSort+MaxLFI have also the same tree size, thus, only the two different tree sizes are reported in Figure 13. LengthSort reduces FP-trees size significantly in all the experiments performed and, consequently, improves the performance. Chain Store data set at a 1 percent support threshold is a special case, where the size of the tree is 0 (Figure 13.f). This is because the upper bound found for the maximum length is one, and thus we only have single frequent itemsets and the construction of the tree is not necessary.





j) Susy

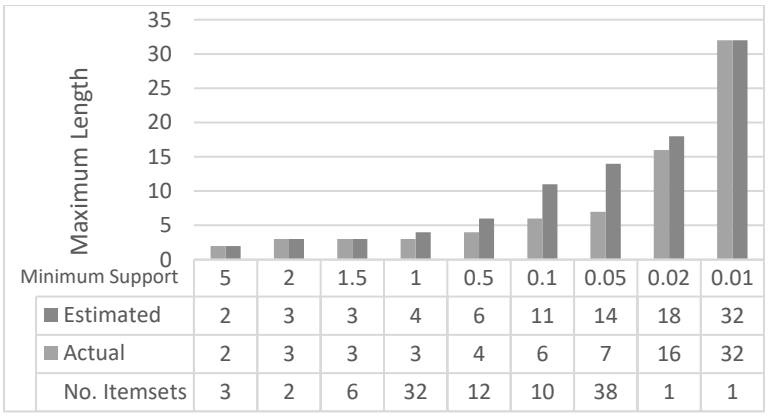
Figure 15: Number of transactions involved in constructing the initial FP-tree

Significant reduction to the size of the initial FP-tree is achieved by pruning both the transactions and the items. Figures 14 compares the number of items in the H-table and Figure 15 gives the number of transactions that are included in the initial FP-tree. The Figures show that, in most cases, only a small fraction of the transactions and/or items are used to construct the initial FP-tree. For LFIMiner and MaxLFI, the number of items in the H-table always increases as the minimum support threshold decreases. This is expected, as only the items whose support is less than the threshold are dropped. However, LengthSort also prunes the items than cannot contribute to the itemsets that are long enough according to the estimated maximum length. For LengthSort, if we compare the number of the included itemsets in the runs, in more than 14 percent of the instances (8 out of 56), the number of items decreased as the support threshold increased.

The number of transactions included in tree construction is constant in LFIMiner and MaxLFI, regardless of the threshold. LengthSort, on the other hand, prunes the transactions and, as depicted in Figure 15, in some instances, the number of transactions is reduced as the support threshold decreases. As an example, consider the extremely low threshold support in the Groceries and BMS-Web-View 1 datasets (Figure15.a and 15.b). We see that only one transaction has survived the pruning. The support count in these two examples is equal to one, so the only element in the LFI is the longest transaction in the dataset. As the longest transaction is unique in both datasets,

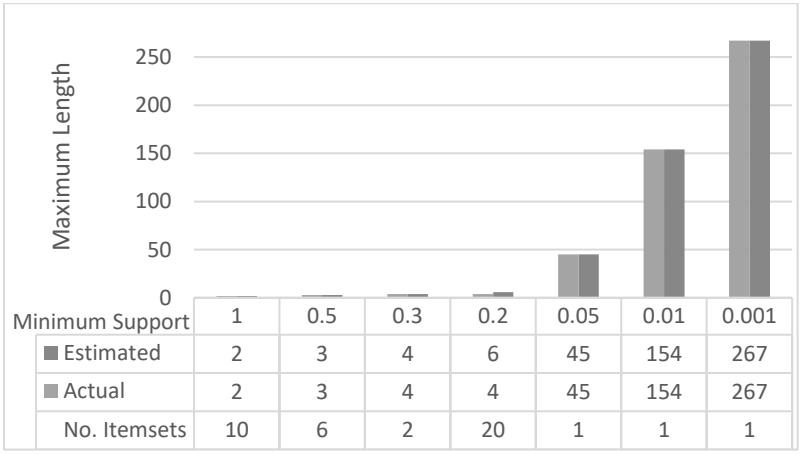
only one transaction is included. This is the reason that in some datasets, the Length-Sort running time decreases as the support threshold decreases. Small support thresholds can increase the maximum length, and as the maximum length increases, more transactions and consequently more items are pruned. Figures 15.i and 15.h, show that the number of included transactions are very similar for Chess and lower support values of Mushrooms. This leads to the lower performance of LengthSort on the two datasets.

Figure 16 gives the estimated maximum length (upper bound), the actual maximum length and the number of itemsets in the LFIs.



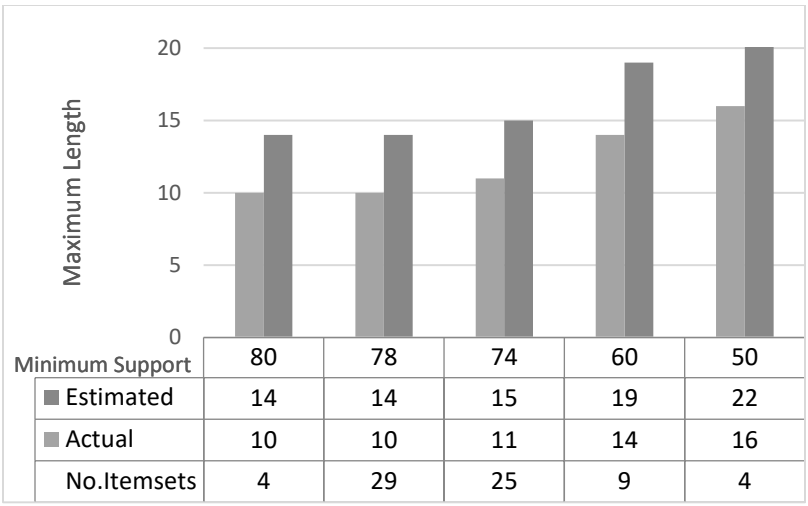
Minimum Support	5	2	1.5	1	0.5	0.1	0.05	0.02	0.01
Estimated	2	3	3	4	6	11	14	18	32
Actual	2	3	3	3	4	6	7	16	32
No. Itemsets	3	2	6	32	12	10	38	1	1

a) Groceries



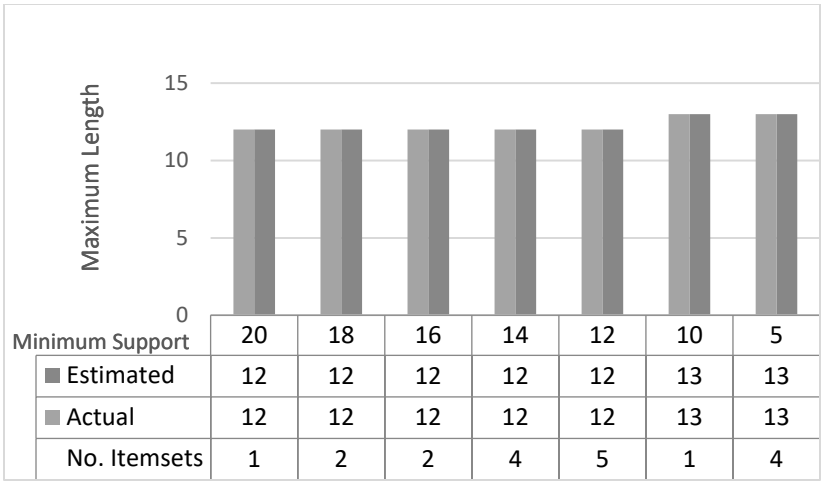
Minimum Support	1	0.5	0.3	0.2	0.05	0.01	0.001
Estimated	2	3	4	6	45	154	267
Actual	2	3	4	4	45	154	267
No. Itemsets	10	6	2	20	1	1	1

b) BMS-Web-View1



Minimum Support	80	78	74	60	50
Estimated	14	14	15	19	22
Actual	10	10	11	14	16
No.Itemsets	4	29	25	9	4

i) Chess



Minimum Support	20	18	16	14	12	10	5
Estimated	12	12	12	12	12	13	13
Actual	12	12	12	12	12	13	13
No. Itemsets	1	2	2	4	5	1	4

j) Susy

Figure 16: Estimated and actual maximum length of the frequent itemsets

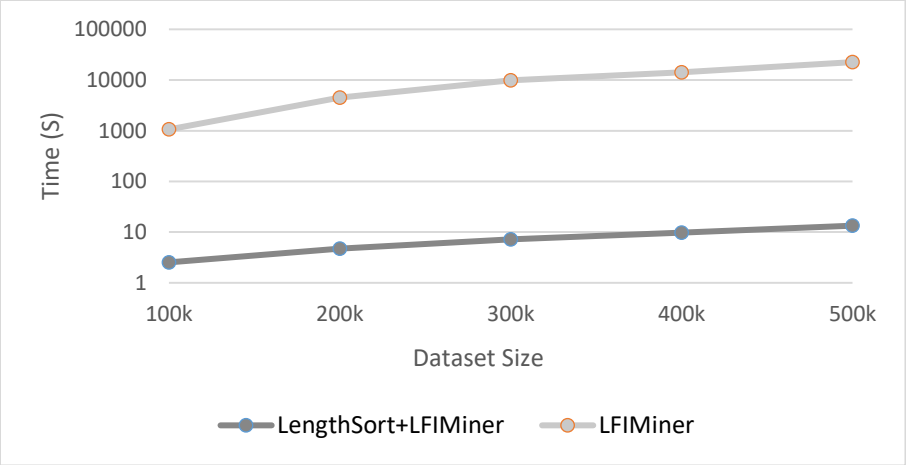
As mentioned, the second run of the LengthSort procedure and the updating to the FP-tree are both needed in the event that the estimated and actual maximum length are not equal. The second run of the LengthSort was necessary in five of the nine experiments performed on the Groceries, four out of six for Mushooms and all cases of the Chess dataset. The maximum difference between the actual and estimated maximum length also occurs in the Groceries dataset (14 vs 7 for a 0.05 percent threshold). When comparing with the other datasets, the ratios of the items that survived the pruning are higher in the Groceries, Mushrooms, Chess and Susy datasets (See Figure 14). So a less successful item pruning may contribute to the inaccurate estimate of the maximum length. It also worth mentioning that these datasets are denser than the other datasets. However, as we can see from Figure 12, LengthSort results in improvements to the run time of the LFI mining process in the most cases when the estimated length was not accurate and an FP-tree update was necessary.

As shown in Figure 16, the estimated maximum length is accurate in most of the experiments carried out on the other datasets. LengthSort decreased the running time in all the instances that the estimated lengths were accurate.

On comparing the number of itemsets in the LFIs with the actual and estimated lengths, an inaccurate estimation of maximum lengths usually occurs when several itemsets are present in the LFI. So we argue that the accuracy of estimation does not depend on the length of the itemsets, but rather on the number of itemsets in the LFI. To summarize, the variation of the transactions' lengths, density of the dataset and the number of itemsets in LFI, are the main influencing factors on the accurate estimation of the length.

Parameters	
No. transactions	100K - 500K
Average length of transactions	10
Number of items	5000

a) synthetic dataset parameters

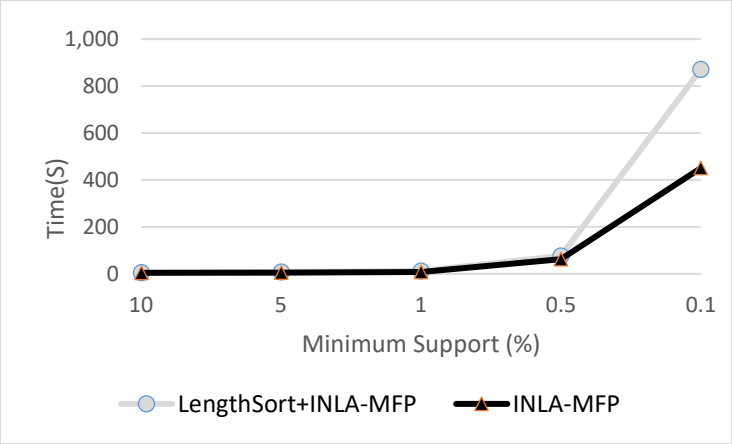


b) scalability test results

Figure 17: Scalability analysis of LengthSort

To test the scalability of the proposed algorithm, five synthetic datasets with different number of transactions were generated using IBM Quest dataset generators (available at [33]). Figure 17 shows the parameters used to create the datasets and the runtime of the algorithms with respect to their dataset size. Except for the number of transactions, all the parameters were kept the same for generating the five datasets. As these parameters are the same, the five datasets are very similar and they differ only in the size. The minimum support threshold was fixed on 0.5 percent during the experiment. The results show that LengthSort achieves good performance in terms of scalability.

As mentioned in section 2, LFI can be mined by post-pruning the output of MFI algorithms. To compare the performance gap, we performed the experiments on a subset of abovementioned datasets. Figure 18 shows the running times of INLA-MFP and LengthSort+INLA-MFP.



d) Chess

Figure 18: Runtime comparisons for INLA-MFP and LengthSort.

As can be seen from Figure 18, INLA-MFP runtimes are greater than the ones of LFIMiner and MaxLFI (see Figure 12) except for 1 percent support in Kosarak dataset. It should be noted that INLA-MFP returns the complete set of MFI while the other two algorithms return LFI which is smaller. Moreover, unlike LFIMiner and MaxLFI, INLA-MFP does not prune the itemsets that are not long enough to be part of LFI. This increases the runtime of the INLA-MFP.

The figure also demonstrates that LengthSort can significantly improve the efficiency of the INLA-MFP except in Chess dataset, however LengthSort+LFIMiner and LengthSort+MaxLFI are still faster (except for 0.5 percent support in Kosarak) in mining LFI. FP-growth based mining algorithms (LFIMinwr and MaxLFI) are less efficient when it comes to sparse datasets [7]. LengthSort in fact makes the datasets denser by pruning the transactions and items. So LengthSort prepares a denser input and is suitable to be coupled with FP-growth based algorithms. LengthSort+ INLA-MFP is useful for mining MFI that are longer than a given threshold.

5 Conclusions

In this study, we developed a novel algorithm called LengthSort, which can efficiently prune datasets to mine the maximum length frequent itemsets. This is especially important because, in many applications, frequent sets with a maximum length are more interesting and many of the frequent sets with lower length are redundant [34]. The novelty of the LengthSort algorithm lies in its feature of using information hidden within the lengths of the transactions. Using this implicit

information helps users to determine the upper bound of the maximum length more efficiently and, furthermore, to prune the unnecessary items and reduce the searching space for the frequent itemsets. The fact that LengthSort partitions the dataset according to its length will also result in reducing the search space by eliminating the unwanted transactions. Thus, by using LengthSort, we can shrink the searching space horizontally by pruning the items, as well as vertically by discarding the unnecessary transactions. Our experimental results achieved on several datasets confirmed that LengthSort pruning reduces the computational effort of mining maximum length frequent itemsets.

Unlike similar algorithms, our proposed pruning methods are applied to dataset before constructing the FP-tree. This is especially beneficial as the mining will then be performed on smaller initial and conditional FP-trees. LengthSort is also less sensitive to the support threshold.

Finally, LengthSort can be applied to mine other type of frequent itemsets. With minimal modifications, LengthSort can be used to mine itemsets longer than a user specified threshold more efficiently. In a similar vein, LengthSort can complement other tree-based algorithms to mine, for instance, long maximal and long closed frequent itemsets in a more effective manner.

The main limitation of the LengthSort algorithm is the fact that it cannot be as effective on dense datasets that have many nominal or ordinal variables represented by dummy binary variables, since, in these cases, the lengths of the most of the transactions are very close to each other. Thus, the upper bound estimates and the pruning process will not be as beneficial. The datasets considered in the experiments were mostly sparse, and further research is needed on LengthSort in order to determine, with a greater level of precision, the kinds of datasets on which it works most efficiently. We expect the algorithm to work more efficiently on datasets with more variations on transactions' lengths. Employing the pruning techniques of LFIMiner and MaxLFI on N-List structure, would also be a fruitful path for future studies. Finally, it would be interesting to apply the LengthSort approach to mining long/maximum length high utility itemsets.

Acknowledgments: This paper is produced as part of the Erasmus Mundus Joint Doctorates (EMJDs) Programme, European Doctorate in Industrial Management (EDIM), funded by the European Commission, Erasmus Mundus Action 1.

Conflicts of Interest: None.

References

[1] R. Agrawal, T. Imieliński, A. Swami, Mining association rules between sets of items in large databases, in: ACM SIGMOD Rec., 1993. http://dl.acm.org/citation.cfm?id=170072 (accessed October 8, 2015).

[2] S. Brin, R. Motwani, C. Silverstein, Beyond market baskets: Generalizing association rules to correlations, ACM SIGMOD Rec. 26 (1997) 265–276. http://dl.acm.org/citation.cfm?id=253327 (accessed February 2, 2016).

[3] E. Baralis, L. Cagliero, T. Cerquitelli, S. Chiusano, P. Garza, Digging deep into weighted patient data through multiple-level patterns, Inf. Sci. (Ny). 322 (2015) 51–71. doi:10.1016/j.ins.2015.06.006.

[4] P.-N. Tan, M. Steinbach, V. Kumar, Association analysis: Basic concepts and algorithms. In Introduction to Data Mining, Pearson Addison Wesley, Boston, 2005.

[5] R. Agrawal, H. Mannila, R. Srikant, Fast Discovery of Association Rules., Adv. Knowl. Discov. Data Mining,. 12 (1996) 307–328. http://www.cs.helsinki.fi/hannu.toivonen/pubs/advances.pdf (accessed October 8, 2015).

[6] J. Han, J. Pei, Y.Y.-A. sigmod Record, U. 2000, Mining frequent patterns without candidate generation, in: ACM SIGMOD Rec., 2000: pp. 1–12. http://dl.acm.org/citation.cfm?id=335372 (accessed November 30, 2017).

[7] Z. Deng, Z. Wang, J. Jiang, A new algorithm for fast mining frequent itemsets using N-lists, Sci. China Inf. Sci. 55 (2012) 2008–2030. doi:10.1007/s11432-012-4638-z.

[8] Y. Djenouri, M. Comuzzi, GA-Apriori: Combining apriori heuristic and genetic algorithms for solving the frequent itemsets mining problem, Inf. Sci. (Ny). 420 (2017) 1–15. http://dx.doi.org/10.1016/j.ins.2017.08.043.

[9] S. Moens, E. Aksehirli, B. Goethals, Frequent Itemset Mining for Big Data, in: Proc. 2013 {IEEE} Int. Conf. Big Data, 2013: pp. 111–118.

[10] B. Mobasher, H. Dai, T. Luo, M. Nakagawa, Effective personalization based on association rule discovery from web usage data, in: Proc. 3rd Int. Work. Web Inf. Data Manag., 2001: pp. 9–15. doi:10.1145/502932.502935.

[11] M.D. Ruiz, M.J. Martin-bautista, D. Sánchez, M.A. Vila, M. Delgado, Anomaly detection using fuzzy association rules, Int. J. Electron. Secur. Digit. Forensics. 6 (2014) 25–37.

[12] B. Vo, S. Pham, T. Le, Z.. Deng, A novel approach for mining maximal frequent patterns., Expert Syst. Appl. 73 (2017) 178–186.

[13] R.C. Agarwal, C.C. Aggarwal, V.V. V. Prasad, Depth first generation of long patterns, in: Proc. Sixth ACM SIGKDD Int. Conf. Knowl. Discov. Data Min., 2000: pp. 108–118. doi:10.1145/347090.347114.

[14] D. Lin, Z.M. Kedem, Pincer-Search : A New Algorithm for Discovering the Maximum Frequent Set 2 Association Rule Mining, in: Int. Conf. Extending Database Technol. Springer, Berlin, Heidelberg., 1998: pp. 103–119.

[15] J. Pei, J. Han, R. Mao, CLOSET:An efficient algorithm for mining frequent closed itemsets in

dynamic transaction databases, in: ACM SIGMOD Work. Res. Issues Data Min. Knowl. Discov., 2000: pp. 21–30. doi:10.1504/IJISTA.2008.017275.

[16] T. Hashem, R.M. Karim, M. Samiullah, C. Farhan, An efficient dynamic superset bit-vector approach for mining frequent closed itemsets and their lattice structure, Expert Syst. Appl. 67 (2017) 252–271. doi:10.1016/j.eswa.2016.09.023.

[17] T. Hu, S.Y. Sung, H. Xiong, F. Qian, Discovery of maximum length frequent itemsets, Inf. Sci. (Ny). 178 (2008) 69–87. doi:10.1016/J.INS.2007.08.006.

[18] J. Friedman, T. Hastie, R. Tibshirani, The elements of statistical learning, Springer, Berlin: Springer series in statistics, New York, 2001. http://statweb.stanford.edu/~tibs/book/preface.ps (accessed October 8, 2015).

[19] T.A. Tai, N.T. Phong, N.K. Anh, An efficient algorithm for discovering maximum length frequent itemsets, in: Third Int. Conf. Knowl. Syst. Eng. IEEE, 2011: pp. 62–69. http://ieeexplore.ieee.org/abstract/document/6063446/ (accessed November 30, 2017).

[20] W. Cheung, Osmar R. Zaiane, Incremental mining of frequent patterns without candidate generation or support constraint, Proc. Seventh Int. Database Eng. Appl. Symp. IEEE. (2003) 111–116. http://ieeexplore.ieee.org/abstract/document/1214917/ (accessed November 30, 2017).

[21] Z. Deng, S. Lv, Expert Systems with Applications PrePost + : An efficient N-lists-based algorithm for mining frequent itemsets via Children – Parent Equivalence pruning, Expert Syst. Appl. 42 (2015) 5424–5432. doi:10.1016/j.eswa.2015.03.004.

[22] J. Singh, H. Ram, Improving efficiency of Apriori algorithm using transaction reduction, Int. J. Sci. Res. Publ. 3 (2013) 1–4.

[23] FP-Growth Association Rule Mining in MATLAB, (n.d.). http://yarpiz.com/98/ypml116-fp-growth (accessed March 7, 2018).

[24] A. Salam, M.S.H. Khayal, Mining top−k frequent patterns without minimum support threshold, Knowl. Inf. Syst. 30 (2012) 57–86. http://www.springerlink.com/index/k48u618rp6676h72.pdf (accessed November 30, 2017).

[25] Z.-H. Deng, DiffNodesets: An efficient structure for fast mining frequent itemsets, Appl. Soft Comput. 41 (2016) 214–223. http://www.sciencedirect.com/science/article/pii/S156849461600017X (accessed November 30, 2017).

[26] S.K. Tanbeer, C.F. Ahmed, B.-S. Jeong, Y.-K. Lee, Efficient single-pass frequent pattern mining using a prefix-tree, Inf. Sci. (Ny). 179 (2009) 559–583. http://www.sciencedirect.com/science/article/pii/S0020025508004532 (accessed November 30, 2017).

[27] J.M.-T. Wu, J. Zhan, J.C.-W. Lin, An ACO-based approach to mine high-utility itemsets, Knowledge-Based Syst. 116 (2017) 102–113. http://www.sciencedirect.com/science/article/pii/S0950705116304191 (accessed December 6, 2017).

[28] M. Hahsler, K. Hornik, T. Reutterer, Implications of probabilistic data modeling for mining association rules, in: M. Spiliopoulou, R. Kruse, C. Borgelt, A. Nurnberger, W. Gaul (Eds.), From Data Inf. Anal. to Knowl. Eng., Springer, 2006: pp. 598–605. http://link.springer.com/content/pdf/10.1007/3-540-31314-1.pdf#page=617 (accessed November 30, 2017).

[29] T. Brijs, G. Swinnen, K. Vanhoof, G. Wets, Using association rules for product assortment decisions: A case study, in: Proc. Fifth Int. Conf. Knowl. Discov. Data Min., San Diego (USA), 1999: pp. 254–260. http://dl.acm.org/citation.cfm?id=312241 (accessed December 6, 2017).

[30] P. Fournier-Viger, J.C.-W. Lin, A. Gomariz, T. Gueniche, A. Soltani, Z. Deng, H.T. Lam, The SPMF Open-Source Data Mining Library Version 2, in: 19th Eur. Conf. Princ. Data Min. Knowl. Discov., Springer, 2016: pp. 36–40. doi:10.1007/978-3-319-46131-1_8.

[31] Frequent Itemset Mining Dataset Repository, (n.d.). http://fimi.ua.ac.be/data/ (accessed March 7, 2018).

[32] SPMF:An Open-Source Data Mining Library, (n.d.). http://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php (accessed March 7, 2018).

[33] IBM Quest Synthetic Data Generator, (n.d.). https://github.com/halfvim/quest (accessed March 7, 2018).

[34] G.I. Webb, Discovering Significant Patterns, Mach. Learn. 68 (2007) 1–33. doi:10.1007/s10994-007-5006-x.