Computing and Informatics, Vol. 37, 2018, 894–914, doi: 10.4149/cai_2018_4_894

AN EFFICIENT ITEMSET REPRESENTATION FOR MINING FREQUENT PATTERNS IN TRANSACTIONAL DATABASES

Savo Tomović, Predrag Stanišić

University of Montenegro
Faculty of Mathematics and Natural Sciences
Džordža Vašingtona bb
81 000 Podgorica, Montenegro
e-mail: {savot, pedjas}@ac.me

Abstract. In this paper we propose very efficient itemset representation for frequent itemset mining from transactional databases. The combinatorial number system is used to uniquely represent frequent k-itemset with just one integer value, for any $k \geq 2$. Experiments show that memory requirements can be reduced up to 300 %, especially for very low minimal support thresholds. Further, we exploit combinatorial number schema for representing candidate itemsets during iterative join-based approach. The novel algorithm maintains one-dimensional array *rank*, starting from $k = 2^{\text{nd}}$ iteration. At the index r of the array, the proposed algorithm stores unique integer representation of the r^{th} candidate in lexicographic order. The *rank* array provides joining of two candidate k-itemsets to be $O(1)$ instead of $O(k)$ operation. Additionally, the *rank* array provides faster determination which candidates are contained in the given transaction during the support count and test phase. Finally, we believe that itemset ranking by combinatorial number system can be effectively integrated into pattern-growth algorithms, that are state-of-the-art in frequent itemset mining, and additionally improve their performances.

Keywords: Frequent itemset mining, Apriori algorithm, combinatorial number system

Mathematics Subject Classification 2010: 68P20, 68P30, 68T30

1 INTRODUCTION

The discovery of frequent itemsets (FI) was introduced in [4] as the first and the most time-consuming phase of the process of finding association rules. The association rule problem is a very important problem in data mining that occupies researchers' attention for decades, with numerous practical applications including market basket analysis, network intrusion detection and pattern discovery in biological applications [8, 9].

An example of association rule from transactional database might be that "85 % of customers who bought *Product X* also bought *Product Y*". Discovering all such rules is important for planning marketing campaigns, designing catalogues, managing prices and stocks, customer relationships management, etc. For example, a shop may decide to place *Product A* close to *Product B* because they are often bought together, to help shoppers finish their task faster. Or the shop may place them at opposite ends of a row, and place other associated items in between to tempt people to buy those items as well, as shoppers walk from one end of the row to the other.

Overall performances of mining association rules are determined by the frequent itemset discovery [5]. Efficient algorithms for generating rules from frequent itemsets can be found in [7]. We do not consider the rule extraction sub-problem in this paper.

Although the pattern growth family of algorithms is considered as state-of-the-art technique in frequent pattern mining, its run-time performance depends on the compaction factor of the data set. For large and sparse databases the performance of the algorithm degrades significantly because it has to generate a very big number of sub-problems and merge the results returned by each sub-problem [7]. In general, the join-based methods are slower but require less memory, while the memory needs of the pattern growth or database projection based algorithms are usually very high, providing that their execution time is smaller [6].

In this paper we introduce the combinatorial number system in context of the frequent itemset mining. With the procedure based on combinatorial numbering schema it is possible to efficiently represent any k-itemset with single integer value. This is "lossless compression", meaning that it is possible to reconstruct the original itemset from the assigned value. Experiments show that memory requirements with such approach can be reduced up to 300 %. It is especially observable for the very low minimal support thresholds when the number of itemsets is the biggest. Usual k-itemset representation requires $O(k)$ integers.

Additionally, we propose Rank Join algorithm which makes several improvements on the Apriori algorithm, that is a classic join-based approach. Although, a hundreds of algorithms have been proposed on various kinds of extensions and applications, the Apriori is still the most commonly recognized reference to evaluate FIM algorithm performances. The Rank Join improves both major phases in Apriori based algorithms: *candidate generation* step and *support count and test* step. Experiments showed that, depending on particular instance of the problem, these two steps can contribute to even more than 90 % of the total execution time [8].

The paper is organized as follows. Section 2 introduces the problem of frequent itemset mining. Related work is reviewed in Section 3. Section 4 introduces combinatorial number system that is base for the new procedure for efficient representation of frequent itemsets. The procedure is named RankItemset and it is presented in Section 5. Section 6 describes a new algorithm named Rank Join, that exploits efficient itemset representation in order to improve both major phases in the Apriori based algorithms. Illustrative example is given in Section 7. Conclusion can be found in Section 8.

2 PRELIMINARIES

This section contains definitions that are necessary for further text. We primarily use notions from [1].

Suppose that $I = \{i_1, i_2, \ldots, i_n\}$ is a finite set; we refer to the elements of I as *items*.

A transaction data set on I is a function $T : \{1, \ldots, |T|\} \to P(I)$. The set $T(k)$ is the k^{th} transaction of T. The numbers $1, \ldots, |T|$ are the transaction identifiers (TIDs).

Given a transaction data set T on the set I, we would like to determine those subsets of I that occur often enough as values of T.

Let $T : \{1, \ldots, |T|\} \to P(I)$ be a transaction data set on a set of items I. The support count of a subset K of the set of items I in T is the number $suppcount_T(K)$ given by: $suppcount_T(K) = |\{t | 1 \leq t \leq |T| \wedge K \subset T(t)\}|$. The support of an itemset K is the number: $support_T(K) = \frac{suppcount_T(K)}{|T|}$.

An itemset K is μ-frequent relative to the transaction data set T if $support_T(K) \geq \mu$. We denote by F_T^μ the collection of all μ-frequent itemsets relative to the transaction data set T and by $F_{T,r}^\mu$ the collection of all μ-frequent itemsets that contain r items for $r \geq 1$.

Note that $F_T^\mu = \bigcup_{r \geq 1} F_{T,r}^\mu$.

Frequent itemset mining problem consists of finding the set F_T^μ for given minimal support μ and transaction data set T.

The following rather straightforward statement is fundamental for the study of frequent itemsets. It is known as Apriori principle.

Let $T : \{1, \ldots, |T|\} \to P(I)$ be a transaction data set on a set of items I. If K and K_1 are two itemsets, then $K_1 \subset K$ implies $support_T(K_1) \geq support_T(K)$.

3 RELATED WORK

During the last two decades numerous algorithms have been proposed to mine frequent itemsets. The interest in this problem still persists [9], mainly due to high computational complexity of the task as well as because it plays an important role in many data mining applications [8].

There are several classifications and consequently several classes of frequent itemsets mining methods [9, 5, 7]. The main challenge of all these approaches is that the number of candidates and frequent itemsets is exponentially large, especially for the lower minimal support thresholds. Although the exponential search space is the fundamental problem of frequent itemset mining, it is possible to significantly speed up the support counting procedure with the use of special memory resident data structures to represent database of transactions [8].

In this section we follow the classification presented in [8, 9]. These papers are the most novel and ones of the most detailed surveys of frequent itemset mining algorithms proposed in the literature.

There are two basic classes of frequent itemset mining algorithms: join-based and tree-based (projection-based) algorithms. The common property of all these algorithms is that they extend prefixes or suffixes of itemsets to generate frequent patterns [8]. Variants of frequent itemset mining, such as maximal [15, 16, 17, 18, 19] and close [20, 21, 22, 23, 24, 25, 26, 27, 28, 29] frequent itemset mining are not considered in this paper.

The join-based algorithms generate candidate itemsets in increasing order of the itemset size, which is usually referred to as level-wise exploration [8, 5]. The candidate generation process of the earliest algorithms used joins. The original Apriori algorithm belongs to this category [8, 9]. It generates $(k+1)$-candidates from previously generated frequent k-itemsets with the use of joins. Then the algorithm computes candidate's support in the database. If the support is equal or higher than the minimum support threshold the itemset is stored as frequent $(k+1)$-itemset. It is important to emphasize that candidate itemsets are generated lexicographically, that is along with pruning irrelevant and duplicate candidates, the key issue of computational efficiency [8, 9].

Although the Apriori is presented as a join-based algorithm, it can be shown that the algorithm is a breadth first exploration of a structured arrangement of the itemsets, known as a set enumeration tree [8]. The set enumeration tree is used to model exponential search space as follows [11]. Given set of items $I = i_1, i_2, \ldots, i_m$ where $i_1 \prec i_2 \prec \ldots \prec i_m$, a set-enumeration is constructed from the root of the tree that represents the empty set. Then the m child nodes of the root representing m 1-itemsets are created. Additionally, for a node representing itemset $i_{j1}i_{j2}\ldots i_{jk}$ and registering i_{jk}, the $(m - jk)$ child nodes representing itemsets $i_{j1}i_{j2}\ldots i_{jk}i_{jk+1}$, $i_{j1}i_{j2}\ldots i_{jk}i_{jk+2}$, $\ldots$, $i_{j1}i_{j2}\ldots i_{jk}i_{jm}$ are created.

Later classes of algorithms explicitly discuss tree-based enumeration [8, 9]. The algorithms from this class can explore the tree using breadth-first, depth-first or hybrid strategies. The breadth-first strategy allows level-wise pruning according to the Apriori principle, that is not possible in the second strategy. The depth-first version allows better memory management because one only needs to maintain a small number of projected transaction sets along the depth of the tree. It is especially desirable to efficiently discover maximal patterns.

The enumeration tree originally is defined on the prefixes of the itemsets. Later algorithms such as FP-Growth find all frequent itemsets ending with a particular suffix by recursively employing a divide-and-conquer strategy to split the problem into smaller sub-problems [5, 7]. It uses the compact data structure called an FP-Tree to build a compressed representation of the input data. The tree is constructed by mapping each transaction onto a path in the FP-Tree. As different transactions can have numerous items in common, their paths may overlap. The compression achieved by using the FP-Tree structure is based on paths' overlaps. If the FP-Tree can be stored in main memory, this will allow to extract frequent itemsets directly from the tree, without additionally passes over the database [7].

Frequent itemsets are generated from the FP-Tree by exploring the tree in a bottom-up fashion. This strategy finds frequent itemsets ending with a particular frequent item and is usually referred to as suffix-based approach [8, 9, 5, 7]. Since every transaction is mapped onto a path in the tree, we can derive the frequent itemsets ending with a particular item, by examining only the paths containing that item – prefix paths. These paths can be accessed efficiently using pointers associated with each node in the tree. Prefix paths are converted into a conditional FP-Tree, which is structurally similar to an FP-Tree, except it is used to find frequent itemsets ending with a particular suffix. FP-Growth uses the conditional FP-Tree for frequent item i to solve the sub-problems of finding frequent itemsets ending in xi where $x \in I \setminus i$.

The size of an FP-Tree is typically smaller than the size of the uncompressed data because it is expected that many transactions often share several items. The best-case scenario is when all transactions have the same set of items and the corresponding FP-Tree contains only a single branch. But, when datasets are sparse, the pattern growth algorithms are inefficient [7, 10]. The worst-case scenario happens when every transaction has a unique set of items. In that case, the size of FP-Tree is higher than the size of the original data [7].

The size of an FP-Tree also depends on how the items are ordered. The usual heuristic is to sort frequent items in decreasing support counts. Nevertheless, ordering by decreasing support counts does not always lead to the smallest tree [7].

Although, the pattern growth family of algorithms is considered as state-of-the-art technique in frequent pattern mining [8], construction of the FP-Tree becomes challenging both from runtime and space complexity as the database grows larger [7, 10]. It is mainly due to the fact that the algorithm has to generate a large number of sub-problems and merge the results returned by each sub-problem. There have been many works that deal with the previous challenges.

In recent years, several data structures, named Node-list [13], N-list [12], Nodeset [11] and DiffNodeset [10] have been presented to enhance the efficiency of mining frequent itemset. They are all based on node sets originated from a prefix tree with encoded nodes. The prefix tree employed by Node-list and N-list uses pre-order number and post-order number to encode each node. The only difference between Node-list and N-list is that Node-list uses descendant nodes to represent an itemset while N-list represents an itemset by ancestor nodes. The Nodeset requires only the pre-order (or post-order) number of a node to encode the node. DiffNodeset only keeps track of differences in the Nodesets of a candidate itemset from its generating frequent itemsets. Compared with the Nodeset, the cardinality of the DiffNodeset is much smaller. Inspired by these data structures several algorithms have been designed. Extensive experimental studies have shown that these algorithms are very effective and usually outperform previous algorithms (FP-Growth∗, Eclat_g) [10].

4 COMBINATORIAL NUMBER SYSTEM

In this section we establish a novel theoretical framework for the frequent itemset mining problem.

Let us review the combinatorial number system and introduce one-to-one correspondence between the integers $1, 2, \ldots, \binom{n}{m}$ and the set of m-combinations of $\{1, 2, \ldots, n\}$ listed in lexicographic order. We primarily use results from [3].

Let $(c_1, c_2, \ldots, c_m)$ represent one such combination where $c_1 < c_2 < \ldots < c_m$. We define

$$complement(n, c_1, c_2, \ldots, c_m) = (d_1, d_2, \ldots, d_m) \tag{1}$$

as the *complement* of $(c_1, c_2, \ldots, c_m)$ with respect to $\{1, 2, \ldots, n\}$, where

$$d_i \leftarrow (n + 1) - c_{m-i+1}. \tag{2}$$

The following function takes n and $(c_1, c_2, \ldots, c_m)$ as input and returns $(d_1, d_2, \ldots, d_m)$ as output in $O(m)$ time [3].

```
**function** COMPLEMENT($n, c_1, c_2, \ldots, c_m$)
Step 1: **for** $i = 1$ **to** $m$ **do**
    $d_i \leftarrow (n + 1) - c_{m-i+1}$
**end for**.
Step 2: COMPLEMENT $\leftarrow (d_1, d_2, \ldots, d_m)$
```

Now, let the *reverse* of $(c_1, c_2, \ldots, c_m)$ be given by $(c_m, c_{m-1}, \ldots, c_1)$. The mapping

$$order(c_1, c_2, \ldots, c_m) = \Sigma_{i=1}^{m} C_i^{c_i-1} \tag{3}$$

has the following properties [3]:

1. if $(c_1, c_2, \ldots, c_m)$ and $(c_1^{'}, c_2^{'}, \ldots, c_m^{'})$ are two m-combinations and the reverse of $(c_1, c_2, \ldots, c_m)$ precedes the reverse of $(c_1^{'}, c_2^{'}, \ldots, c_m^{'})$ in lexicographic order, then
$$order(c_1, c_2, \ldots, c_m) < order(c_1^{'}, c_2^{'}, \ldots, c_m^{'}); \tag{4}$$

2. $order(1, 2, \ldots, m) = 0$ and $order(((n - m + 1)(n - m + 2) \ldots n)) = \binom{n}{m} - 1$ implying that the transformation *order* maps the $\binom{n}{m}$ different m-combinations onto $\{0, 1, \ldots, \binom{n}{m} - 1\}$ while preserving reverse lexicographic order.

The following function takes $(c_1, c_2, \ldots, c_m)$ as input and returns $order(c_1, c_2, \ldots, c_m)$ as output in $O(m^2)$ time [3].

```
function ORDER($c_1, c_2, \ldots, c_m$)
Step 1: $sum \leftarrow 0$
Step 2: for $i = 1$ to $m$ do
            $sum \leftarrow sum + \binom{c_i - 1}{i}$
        end for.
Step 3: ORDER $\leftarrow sum$
```

Using order and complement, we can define the following one-to-one mapping of the set of $\binom{n}{m}$ possible combinations onto $\{1, 2, \ldots, \binom{n}{m}\}$ which preserves lexicographic ordering:

$$rankc(n, c_1, c_2, \ldots, c_m) = \binom{n}{m} - order(complement(n, c_1, c_2, \ldots, c_m)).$$

Thus $rankc(n, 1, 2, \ldots, m) = 1$, $rankc(n, 1, 2, \ldots, m, m+1) = 2$, ..., $rankc(n, (n-m+1), (n-m+2), \ldots n) = \binom{n}{m}$. The following procedure is an implementation of the preceding mapping: it takes n and the combinations $(c_1, c_2, \ldots, c_m)$ as input and returns the ordinal position h of the later in $O(m^2)$ time [3].

```
procedure RANKC($n, c_1, c_2, \ldots, c_m, h$)
Step 1: $h \leftarrow C_m^n$
Step 2: $(d_1, d_2, \ldots, d_m) \leftarrow COMPLEMENT(n, c_1, c_2, \ldots, c_m)$
Step 3: $h \leftarrow h - ORDER(d_1, d_2, \ldots, d_m)$
```

Let us explain the complexity of the RANKC in more details. In step 2 it calls function COMPLEMENT with $O(m)$ complexity. But, the dominant step is 3rd in which function ORDER is called. Complexity for ORDER is $O(m^2)$ as we stated earlier. Actually, in the second step in ORDER function there is loop m iterations long, where in each iteration $\binom{c_i - 1}{i}$ is calculated, $1 \leq i \leq m$. Upper bound complexity for each iteration is $O(m)$, so overall complexity for ORDER is $O(m^2)$.

Here we will just mention the question of inverting the RANKC mapping. Specifically, given an integer h, where $1 \leq h \leq \binom{n}{k}$, it is required to determine the combination $(c_1 c_2 \ldots c_k)$ such that $rankc(n, c_1, c_2, \ldots, c_k) = h$. The function ORDERINV$(n, k, g)$ [3] is an implementation of the mentioned mapping. It takes n, k and h as input and returns a combination $(c_1 c_2 \ldots c_k)$ as output in $O(nk)$ time.

In the end of this section we propose hypothetical algorithm – Apriori Combinatorial for mining frequent itemsets in previously defined framework.

We use two dimensional array *frequentItemsets* to store support counts for k-combinations that appear in database. In the iteration k all frequent k-combinations will be generated. Instead of explicit candidate generation, our algorithm uses combinatorial number schema to map k-combinations to indexes of the array *frequentItemsest*$[k-1]$. This mapping preserves lexicographical ordering, so combination $(1, 2, \ldots, k)$ is mapped to 0 and appropriate support is stored in *frequentItemsets*$[k-1][0]$. Similarly, combination $c_1c_2, \ldots, c_k$ is mapped to $r = RANKC(n, c_1, c_2, \ldots, c_k)$ while corresponding support count is stored in *frequentItemsets*$[k-1][r]$.

In the support count phase in the iteration k, each transaction is read, and the support for all k-combinations contained in the transaction is incremented.

Let us compare complexity of the Apriori Combinatorial to the original Apriori algorithm. We have to compare complexities of the two main steps: candidate generation step and support count step.

In the Apriori Combinatorial algorithm, there is no candidate generation step. Actually, in the iteration k, it is sufficient to allocate $\binom{n}{k}$ integers to the array *frequentItemsets*$[k-1]$ and initialize them to 0.

For support counting, in each iteration the algorithm reads all transactions from the database. The algorithm computes $RANKC(s), s \subset t, |s| = k$ and increments *frequentItemsets*$[k-1][RANKC(s)]$. As it will be explained in the next section, by using Pascal's triangle it is possible to reduce complexity for $RANKC$ to $O(k)$. Consequently, the complexity for support counting phase in the Apriori Combinatorial is $O\left(|T|\Sigma_k k\binom{t_{max}}{k}\right)$, where t_{max} is the size of the longest transaction and $|T|$ is the number of transactions in the database. Notice that factor α_k that appears in complexity estimation for the original Apriori is eliminated.

The previous considerations are summarized in Table 1.

	Apriori-Like Algorithms	Apriori Combinatorial
candidate generation step	$\Sigma_{k=2}^{w}(k-1)\lvert F_{k-1}\rvert^2$	$O(1)$
creating hash tree	$\Sigma_{k=2}^{w}k\lvert C_k\rvert$	0
candidate prunning	$\Sigma_{k=2}^{w}k^2\lvert C_k\rvert$	0
support counting	$O\left(\lvert T\rvert\Sigma_k \alpha_k\binom{t_{max}}{k}\right)$	$O\left(\lvert T\rvert\Sigma_k k\binom{t_{max}}{k}\right)$

Table 1.

As we mentioned earlier, the Apriori Combinatorial is a hypothetical algorithm, because it is impossible to manage the array *frequentItemsets* in cases where memory capacity is limited, as it is expected to be. But still, it is possible to implement the ideas from the Apriori Combinatorial, and significantly improve both major steps in a join-based algorithms. In the next section we propose novel approach for efficient itemset representation. In Section 6 we present Rank Join algorithm that outperforms other Apriori-like algorithms by efficient implementation of the candidate generation and support count steps.

5 EFFICIENT ITEMSET REPRESENTATION

In this section we give a short introduction to the join-based approach and review the Apriori algorithm from [4]. The Apriori is famous and widely-used algorithm for the frequent itemset mining.

The Apriori algorithm iteratively generates frequent itemsets starting from frequent 1-itemsets to frequent itemsets of the maximal size. Each iteration of the Apriori consists of two phases: *candidate generation* and *support count and test.*

In the candidate generation phase potentially frequent k-itemsets or candidate k-itemsets are generated. The anti-monotone property of the itemset support is used in this phase and it provides elimination or pruning of some candidates without calculating its actual support (candidate containing at least one not frequent subset is pruned immediately, before support counting phase according to the Apriori principle).

The set C_k, that contains candidate k-itemsets, is generated from F_{k-1}. The set F_{k-1} is known from the previous iteration and contains frequent $(k-1)$-itemsets. The two $(k-1)$-frequent itemsets $c_1c_2\dots c_{k-1}$ and $d_1d_2\dots d_{k-1}$ give candidate k-itemset e if and only if $c_1 = d_1 \wedge c_2 = d_2 \wedge \dots \wedge c_{k-2} = d_{k-2}$ and $c_{k-1} < d_{k-1}$. In that case, we have $e = c_1c_2\dots c_{k-1}d_{k-1}$.

The support count and test phase consists of calculating support counts for all previously generated candidates (the set C_k) and tests the counts to the minimal support threshold. In this phase, it is essential to efficiently determine if the candidates are contained in a particular transaction in order to increment their support. Candidates that have sufficiently large support count, are termed as frequent itemsets and they are stored as elements of the F_k.

The Apriori algorithm terminates when none of the frequent itemsets can be generated.

We will use $C_{T,k}$ to denote candidate k-itemsets which are generated in the iteration k over the transactional database T. By $F^{\mu}_{T,k}$ we denote k-frequent itemsets in the database T, having support count $\geq \mu$, where μ is minimal support threshold. If T and μ are known from the context, we will omit them. Pseudo-code for the Apriori algorithm comes next.

```
Algorithm:    Apriori
Input:  A transaction dataset $T$;
Input:  Minimal support threshold $\mu$
Output:  $F^{\mu}_{T}$
Method:
    1.  $C_{T,1} = \{\{i\} | i \in I\}$
    2.  $F^{\mu}_{T,1} = \{K \in C_{T,1} | supp_T(K) \geq \mu\}$
    3.  for ($k$=2; $F^{\mu}_{T,k-1} \neq \emptyset$; k++)
            $C_{T,k} = \text{apriori\_gen}(F^{\mu}_{T,k-1})$
            $F^{\mu}_{T,k} = \{K \in C_{T,k} | supp_T \geq \mu\}$
    end for
4.  $F^{\mu}_{T} = \bigcup_{r\geq 1} F^{\mu}_{T,r}$
```

Let us now introduce the procedure that will reduce memory requirements for storing the set of all μ-frequent k-itemsets $F^{\mu}_{T,k}$ in each iteration. Consequently, the size of the final set $F^{\mu}_{T} = \bigcup_{r\geq 1} F^{\mu}_{T,r}$ that contains all μ-frequent itemsets in the given database is reduced.

The idea is to represent each k-frequent itemset with just one integer. The following procedure named *RankItemset* is an implementation of the mapping: it takes k-itemset $(c_1, c_2, \ldots, c_k)$ as an input and returns the ordinal position h of the later. The procedure is based on results from [3], here reviewed in the previous section.

```
procedure RankItemset($c_1, c_2, \ldots, c_k, h$)
Step 1: $sum \leftarrow 0$
Step 2: for $i = 1$ to $k$ do
            $sum \leftarrow sum + \binom{n-c_{k-i+1}}{i}$
        end for
Step 3: $h \leftarrow C^n_k - sum$
```

The *RankItemset* preserves lexicographic ordering. Thus $\mathit{RankItemset}(1, 2, \ldots, k) = 1, \mathit{RankItemset}(1, 2, \ldots, k+1) = 2, \ldots, \mathit{RankItemset}((n-k+1), (n-k+2), \ldots n) = \binom{n}{k}$, where n is the number of items.

Let us estimate the complexity of the *RankItemset*. The dominant step is the second one in which *sum* is calculated. There is a loop k iterations long, where in each iteration $\binom{n-c_{k-i+1}}{i}$ is calculated, $1 \leq i \leq k$. Upper bound complexity for each iteration is $O(k)$, so the overall complexity is $O(k^2)$.

It is possible to improve the *RankItemset* if we compute all $\binom{n-c_{k-i+1}}{i}$ in advance. We have $1 \leq c_i \leq n$ and $1 \leq i \leq k$, so we can generate Pascal's Triangle with n rows and k columns. The Pascal's triangle can be stored as lower triangle matrix, where element $(i, j), j < i$ is $\binom{i}{j}$.

Using the Pascal's Triangle we reduce complexity to $O(k)$; there is still a loop k iterations long, where in each iteration $\binom{n-c_{k-i+1}}{i}$ is calculated, $1 \leq i \leq k$, but now in $O(1)$ time. The previous implies that the overall complexity of the *RankItemset* is reduced to $O(k)$. Similarly, the complexity of the *RANKC* from the previous section can be reduced to $O(k)$, as we stated earlier.

The RankItemset procedure allows to change the step $F^{\mu}_{T,k} = \{K \in C_{T,k} | supp_T \geq \mu\}$ with the following $F^{\mu}_{T,k} = \{\mathit{RankItemset}(K, h_K) | K \in C_{T,k} \wedge supp_T(K) \geq \mu\}$. In other words, instead of storing each frequent itemset explicitly, we store corresponding *RankItemset* value. In that way we reduce space requirements for k-itemset storing to $O(1)$ instead of $O(k)$. At the same time, the previous procedure does not degrade time complexity because the *RankItemset* is $O(k)$ as well as making a copy of k-itemset and storing it in $F^{\mu}_{T,k}$.

Here we will just mention that the function ORDERINV(n, k, g) [3] is an implementation of inverting the *RankItemset*. It takes n, k and h as an input and returns an itemset $(c_1c_2 \dots c_k)$ as an output in $O(nk)$ time. In other words, the proposed representation of the itemsets does not influence the final result, i.e., we always obtain the same set of the frequent patterns.

It is possible to integrate the *RankItemset* procedure into the most efficient pattern-growth algorithm [14, 13] and improve their space efficiency. Additionally, candidate ranking can be used to encode nodes in the set enumeration tree instead of pre-order or/and post-order number. Node encoding can be done along with the tree construction procedure, providing that several pre-order or/and post-order tree traversals are not necessary any more.

6 RANK JOIN ALGORITHM

In this section we propose the Rank Join algorithm for frequent itemset mining. It implements ideas from the previous sections and improves, at the first place, the *candidate generation* procedure, but also the *support count and test* step.

The Rank Join maintains one-dimensional array *rank*, starting from $k = 2^{\text{nd}}$ iteration. At the index r of the array, the algorithm stores the *RankItemset* of the r^{th} candidate in the lexicographic order, so $\textit{rank}[r] = \textit{RankItemset}(c_1^r, c_2^r, \dots, c_k^r)$, where $c_1^r c_2^r \dots c_k^r$ is the r^{th} candidate in lexicographic order in the iteration k.

The array *rank* is used for efficient implementation of "joining" procedure in the candidate generation phase as follows.

As it is described in the previous section, in the iteration k, a join-based algorithm generates candidate k-itemsets from frequent $(k-1)$-itemsets (known from the previous iteration). More precisely, two $(k-1)$-frequent itemsets $c_1c_2 \dots c_{k-1}$ and $d_1d_2 \dots d_{k-1}$ give candidate k-itemset e if and only if $c_1 = d_1 \wedge c_2 = d_2 \wedge \dots \wedge c_{k-2} = d_{k-2}$ and $c_{k-1} < d_{k-1}$. In that case, we have $e = c_1c_2 \dots c_{k-1}d_{k-1}$. So, each "joining" is of $O(k)$ complexity.

Having in hand the *rank* array, the Rank Join will join $c_1c_2 \dots c_{k-1}$ and $d_1d_2 \dots d_{k-1}$ in $O(1)$. Actually, let $c_1c_2 \dots c_{k-1}$ be the r^{th} and let $d_1d_2 \dots d_{k-1}$ be the p^{th} frequent $(k-1)$-itemset in lexicographic order. So, in order to check joining condition it is sufficient to test if $\textit{rank}[r] = \textit{rank}[p]$ and $c_{k-1} < d_{k-1}$, which is $O(1)$ operation.

We emphasise here, that the Rank Join requires $O(k)$ time to calculate a rank for one candidate k-itemset. It is compensated in the general join-base algorithm by the step where candidate k-itemset is inserted in a tree structure in $O(k)$ time. The Rank Join does not create tree for storing candidates. The role of the tree in the support counting phase is given to the *rank* array, as will be described later.

Finally, we can conclude that the candidate generation step in the iteration k in a general join-base algorithm is $O((k-1)|F_{k-1}|^2)$, while in the Rank Join it is $O(|F_{k-1}|^2)$.

The *rank* array is used in the support count phase in order to reduce number of candidate itemsets that are compared against transactions from the database. Let us explain the idea in case of the iteration k. All candidates are lexicographically generated and stored in the array C_k. Also, for each candidate $C_k[i]$ holds $rank[i] = RankItemset(C_k[i])$. In order to find support counts for candidate itemsets it is necessary, for each transaction $t \in T$, to enumerate those candidates that are contained in t and consequently increment their support.

Consider transaction $t = t_1 t_2 \dots t_l$. If $l < k$ there is no k-candidate that is contained in t. If $l \geq k$, the lexicographically smallest k-itemset contained in t is $t_1 t_2 \dots t_k$. At the same time, the lexicographically largest k-itemset contained in t is $t_{l-k+1} \dots t_{l-1} t_l$. So, candidates possibly contained in t are $C_k[i]$, where

$RankItemset(t_1, t_2, \dots, t_k) \leq rank[i] \leq RankItemset(t_{l-k+1}, \dots, t_{l-1}, t_l)$.

Because of lexicographical ordering they are placed consecutively. The idea is illustrated in Figure 1. Notice that the *rank* array does not contain NULL pointers.

The previous procedure is similar but more efficient than the one implemented in algorithms from [15] and [14], that is the most efficient join-based algorithms [8]. The later exploits an array $PREFIX_k[]$ according to the following consideration. The various k-itemsets of C_k are stored in a vector lexicographically, and all of them sharing a common 2-item prefix are stored in a continuous section of this vector. Each entry in $PREFIX_k[]$ represents different 2-item prefix, and contains the pointer to the first candidate in C_k characterized by that prefix. When transaction t is processed, the algorithm generates all the possible prefixes of length 2. Once a prefix $\{t_{i1}, t_{i2}\}$ is selected, the possible completions of this prefix needed to build k-subsets of t can only be found in $\{t_{i2+1}, \dots, t_{|t|}\}$. The previous is illustrated in Figure 1.

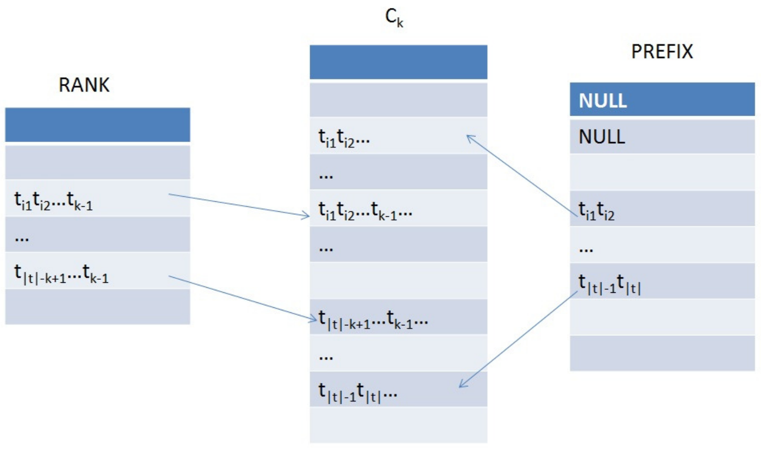


Figure 1. Locating candidates using RANK and PREFIX [15, 14] arrays

The Rank Join is formalized in the following listing.

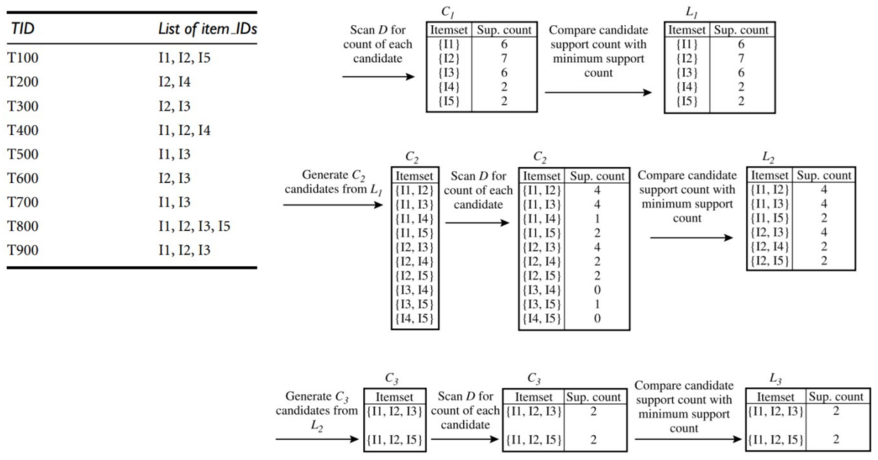
```
Algorithm:    Rank Join
Input:  A transaction dataset $T$; Minimal support threshold $\mu$
Output:  $F_T^\mu$
Method:
    1.  $C_{T,1} = \{\{i\}|i \in I\}$
    2.  $F_{T,1}^\mu = \{K \in C_{T,1}|supp_T(K) \geq \mu\}$
        $rank[i] = i, 1 \leq i \leq |F_{T,1}^\mu|$
    3.  for ($k$=2; $F_{T,k-1}^\mu \neq \emptyset$; k++)
            $C_{T,k} = \{e_1, e_2, \ldots, e_k|\exists c^r, d^p \in F_{T,k-1}^\mu,$
            $c^r = e_1e_2 \ldots e_{k-2}e_{k-1} \wedge d^p = e_1e_2 \ldots e_{k-2}e_k$
             $\wedge\ rank[r] = rank[p]\}$
            for each $t \in T$ do
                $start = RankItemset(t_1, t_2, \ldots, t_k)$
                $end = RankItemset(t_{l-k+1}, , \ldots, t_{k-1}, t_k)$
                for each $C_k[i], start \leq i \leq end$ do
                        if $C_k \subset t\ supp_T(C_k[i]) + +$
                    end for each
            end for each
            $F_{T,k}^\mu = \{K \in C_{T,k}|supp_T(K) \geq \mu\}$
        end for
    4.  $F_T^\mu = \bigcup_{r \geq 1} F_{T,r}^\mu$
```

7 EXAMPLE

In this section, we will illustrate main benefits of the itemset representation with combinatorial number system. Consider transaction dataset T from Figure 2. It contains nine transactions. Generation of the candidate itemsets and frequent itemsets in the original Apriori algorithm, where the minimum support count is 2 transactions, is also illustrated in Figure 2 [5].

There are $6+6+2$ frequent itemsets. In the original Apriori method, k-itemset is represented with k integers. It means, that the Apriori needs $6*1+6*2+2*3 = 24$ integers to store all frequent itemsets.

With the *RankItemset* procedure from Section 5, we can uniquely represent each frequent itemset with just one integer. It takes k-itemset $(c_1, c_2, \ldots, c_k)$ as an input and returns the ordinal position h of the later. For example, the itemset $(1, 2)$ is represented with 1, because it is lexicographically the first 2-combination of $\{1, 2, 3, 4, 5\}$. Similarly, the itemset $(1, 5)$ is represented with 4, while 3 corresponds to the itemset $(1, 2, 5)$ as it is the third 3-combination of $\{1, 2, 3, 4, 5\}$. In the end, we have $6 + 6 + 2 = 14$ frequent itemsets and we use 14 integers to store them. Again, we emphasize that the proposed representation of the frequent itemsets does not influence the final result, i.e., we always obtain the same set of frequent patterns but in the most compact form. It is guaranteed by the combinatorial number system


Scan *D* for count of each candidate
Compare candidate support count with minimum support count
Generate C_2 candidates from L_1
Generate C_3 candidates from L_2

TID	List of item_IDs
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3

C_1

Itemset	Sup. count
{I1}	6
{I2}	7
{I3}	6
{I4}	2
{I5}	2

L_1

Itemset	Sup. count
{I1}	6
{I2}	7
{I3}	6
{I4}	2
{I5}	2

C_2

Itemset
{I1, I2}
{I1, I3}
{I1, I4}
{I1, I5}
{I2, I3}
{I2, I4}
{I2, I5}
{I3, I4}
{I3, I5}
{I4, I5}

C_2

Itemset	Sup. count
{I1, I2}	4
{I1, I3}	4
{I1, I4}	1
{I1, I5}	2
{I2, I3}	4
{I2, I4}	2
{I2, I5}	2
{I3, I4}	0
{I3, I5}	1
{I4, I5}	0

L_2

Itemset	Sup. count
{I1, I2}	4
{I1, I3}	4
{I1, I5}	2
{I2, I3}	4
{I2, I4}	2
{I2, I5}	2

C_3

Itemset
{I1, I2, I3}
{I1, I2, I5}

C_3

Itemset	Sup. count
{I1, I2, I3}	2
{I1, I2, I5}	2

L_3

Itemset	Sup. count
{I1, I2, I3}	2
{I1, I2, I5}	2

Figure 2. Generation of candidate itemsets and frequent itemsets

that introduces one-to-one correspondence between the integers $1, 2, \ldots, \binom{n}{m}$ and the set of m-combinations of $\{1, 2, \ldots, n\}$ listed in lexicographic order.

Finally, we explain how two candidates are joined in the Rank Join in $O(1)$ having in hand the *rank* array. Consider the itemset $(1, 2, 3, 5)$, that will be pruned by the Apriori principle because it contains the subsets that are not frequent (because of that it does not appear in Figure 2). It is generated by joining $(1, 2, 3)$ and $(1, 2, 5)$. In the original Apriori this joining requires three comparisons: $1 = 1$, $2 = 2$ and $3 < 5$. The Rank Join from Section 6, needs just two comparisons $rank(1, 2) = rank(1, 2)$ and $3 < 5$. In general, in the original Apriori algorithm joining of two k-itemsets requests k comparisons, comparing to just 2 in the Rank Join.

8 PERFORMANCE EVALUATION

In order to prove theoretical considerations from Sections 5 and 6 we performed a number of experiments.

Datasets used in experiments are synthetic datasets generated with IBM Quest Data Generator. At the command-line we run *seq_data_generator lit -ascii -ntrans XX -tlen YY -nitems ZZ -fname TXXLYYNZZ*. It will produce file TXXLYYNZZ with XX × 1 000 transactions involving YY average number of items per transaction, drawn from ZZ × 1 000 total number of items. Each line of the file is a transaction. The items in each transaction are represented by item numbers and are separated by spaces. For example, file T1000L10N1 contains 1 000 000 transactions made from 1 000 items, with average length of 10.

We measured memory needed for storing frequent itemsets in KB with respect to *minsup* parameter. Achieved improvements are up to 300 %. We emphasize the fact that the most significant enhancements are achieved for the smallest values of *minsup*, when the number of frequent itemsets is the biggest. Results are presented in Figures 3 and 4.

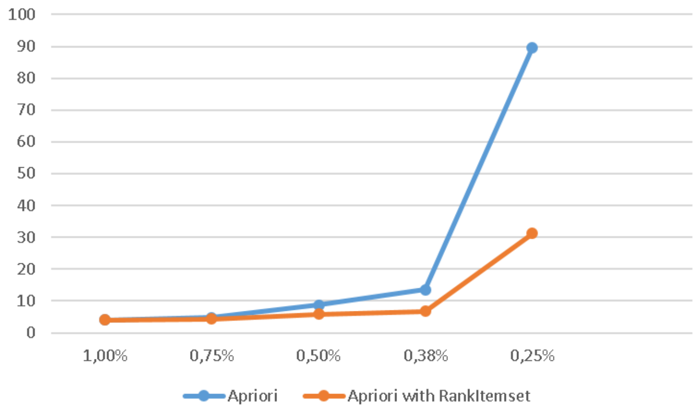


Figure 3. Experiment 1, datasets T200L10N1

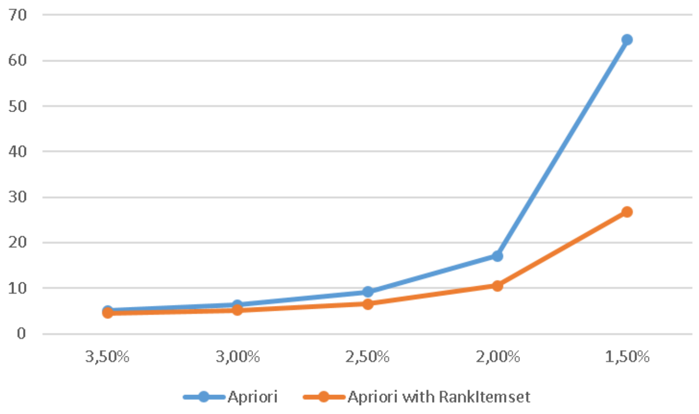


Figure 4. Experiment 1, datasets T100L40N1

We implemented the Rank Join and the original Apriori algorithm with direct count procedure [9], that is the most efficient join-based algorithm [8]. We used programming language C and machine with Intel Celeron 2 GHz and 2 GB of RAM. Datasets used in experiments are synthetic datasets generated with IBM Quest Data Generator as it is already explained.

We measured execution time in seconds with respect to *minsup* parameter. Achieved improvements are between 3 % and 15 %. We emphasize the fact that

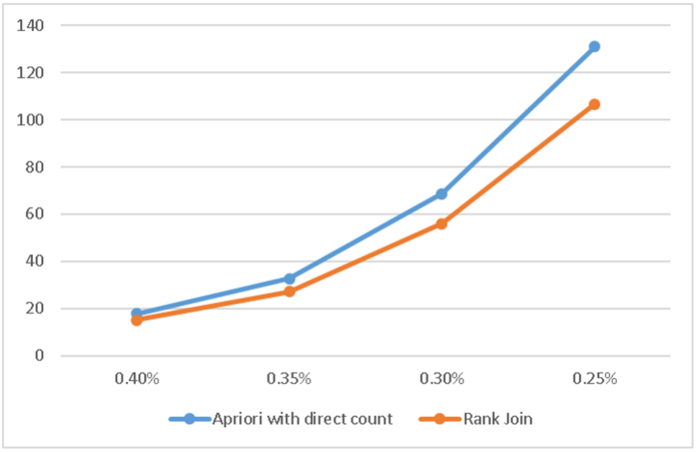


Figure 5. Experiment 2, datasets T100L10N1

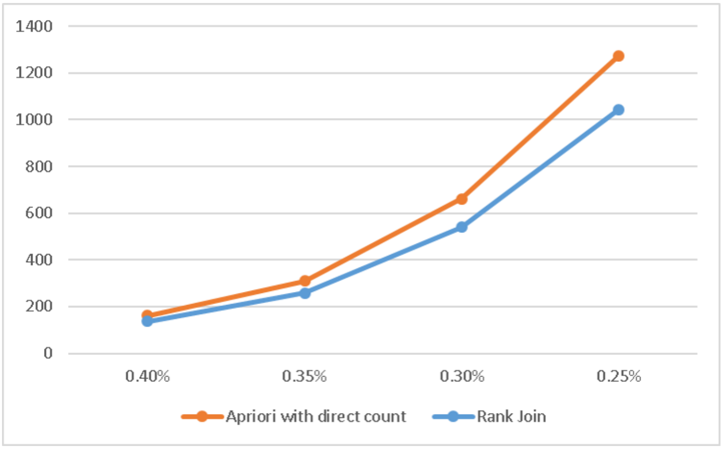


Figure 6. Experiment 2, dataset T1000L10N1

the most significant enhancements are achieved for the smallest values of *minsup*, when the number of candidate itemsets is the biggest. Results of the experiments are presented in Figures 5 and 6 for synthetic datasets and in Figure 7 for Extended BAKERY dataset. The dataset contains information about one year worth of sales information for a couple of small bakery shops. The sales are made by employees. The dataset contains information about the different store locations in West Coast states (California, Oregon, Arizona, Nevada), the assortments of baked goods offered for sale and the purchases made [30]. The experiment shows that the proposed algorithm outperforms the original Apriori on the real dataset, too.

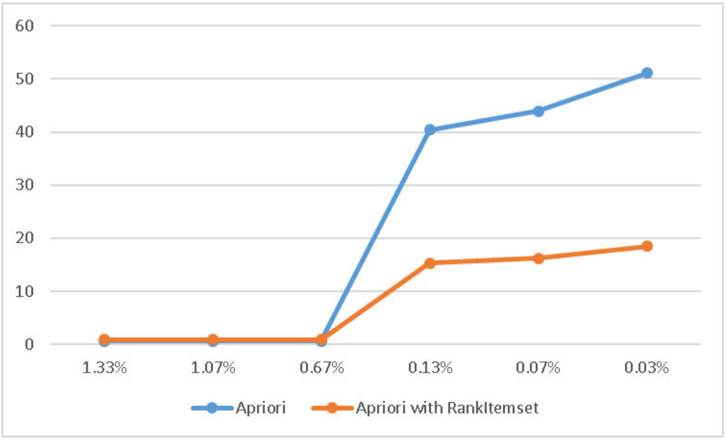


Figure 7. Experiment 2, BAKERY dataset

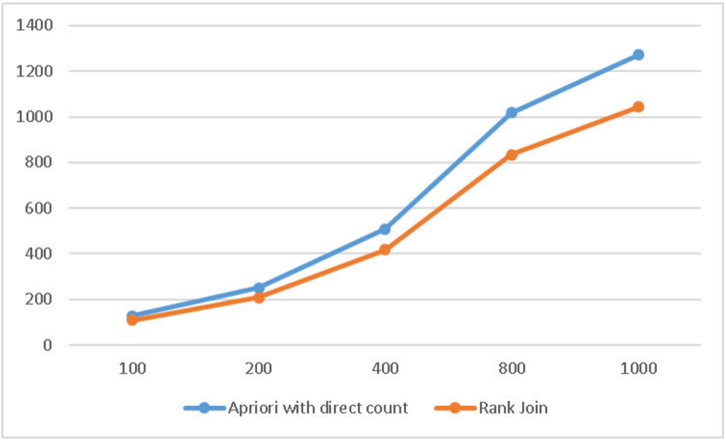


Figure 8. Experiment 3, scalability

In all experiments, the parameter *minsup* is changed in a way that indicates performance differences the best. We started with the greatest *minsup* value for which there was a significant difference in the algorithms' performances and reduced it to the smallest value for which the original Apriori method can finish without "out of memory error".

In Figure 8 we present results of experiment in which Rank Join shows better scalability. We set $minsup = 0.25\,\%$ and change number of transactions from 100 K to 1 000 K.

9 CONCLUSIONS AND FUTURE WORK

In this paper we have proposed a novel procedure for efficient representation of k-itemset with just one integer value. In this segment it is superior by comparison with any other approach. An itemset c is represented with the $RankItemset(c)$ value.

Additionally, we presented new join-based approach called Rank Join. The Rank Join significantly improves both major steps in join-base algorithms. We performed a series of experiments and measured execution times with respect to *minsup* parameter. In all test cases Rank Join is more efficient than any other join-base algorithm. This is especially case for very small *minsup* values when these algorithms generate the biggest number of candidate itemsets.

We believe that candidate ranking by combinatorial number system can be effectively integrated into pattern-growth algorithms, that are state-of-the-art in frequent itemset mining, and additionally improve their performances. As future work, we plan to integrate candidate ranking into the most efficient pattern-growth algorithm [14, 13]. It can improve their efficiency, because candidate ranking can be used to encode nodes in the set enumeration tree instead of pre-order or/and post-order number. Node encoding can be done along with the tree construction procedure, providing that several pre-order or/and post-order tree traversals needed in [10, 11, 12, 13], are not necessary.

REFERENCES

[1] Simovici, D. A.—Djeraba, C.: Mathematical Tools for Data Mining – Set Theory, Partial Orders, Combinatorics. Springer, London, 2008.

[2] Maurer, B. S.—Ralston, A.: Discrete Algorithmic Mathematics. A. K. Peters, Massachusetts, 1998.

[3] Akl, G. S.: The Design and Analysis of Parallel Algorithms. Prentice Hall, New Jersey, 1989.

[4] Agrawal, R.—Imielinski, T.—Swami, A. N.: Mining Association Rules Between Sets of Items in Large Databases. Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93), Washington DC, USA, 1993, pp. 207–216, doi: 10.1145/170035.170072.

[5] Han, J.—Cheng, H.—Xin, D.—Yan, X.: Frequent Pattern Mining: Current Status and Future Directions. Data Mining and Knowledge Discovery, Springer, Vol. 15, 2007, No. 1, pp. 55–86, doi: 10.1007/s10618-006-0059-1.

[6] Ivancsy, R.—Vajk, I.: Automata Theory Approach for Solving Frequent Pattern Discovery Problem. International Journal of Computer, Control, Quantum and Information Engineering, 2007, pp. 203–208.

[7] Tan, P. N.—Steinbach, M.—Kumar, V.: Introduction to Data Mining. Springer, London, 2006.

[8] Aggarwal, C. C.—Bhuiayn, M. A.—Mohammad, H. A.: Frequent Pattern Mining Algorithms: A Survey. In: Aggarwal, C., Han, J. (Eds.): Frequent Pattern Mining. Springer International Publishing Switzerland, 2014, pp. 19–64.

[9] Aggarwal, C. C.: An Introduction to Frequent Pattern Mining. In: Aggarwal, C., Han, J. (Eds.): Frequent Pattern Mining. Springer International Publishing Switzerland, 2014, pp. 1–17, doi: 10.1007/978-3-319-07821-2_1.

[10] Deng, Z. H.: DiffNodesets: An Efficient Structure for Fast Mining Frequent Itemsets. Applied Soft Computing, Vol. 41, 2016, pp. 214–223, doi: 10.1016/j.asoc.2016.01.010.

[11] Deng, Z. H.—Lv, S. H.: PrePost+: An Efficient N-Lists-Based Algorithm for Mining Frequent Itemsets via Children-Parent Equivalence Pruning. Expert Systems with Applications, Vol. 42, 2015, No. 13, pp. 5424–5432.

[12] Deng, Z. H.—Wang, Z. H.—Jiang, J. J.: A New Algorithm for Fast Mining Frequent Itemsets Using N-Lists. Science China Information Sciences, Vol. 55, 2012, No. 9, pp. 2008–2030.

[13] Deng, Z. H.—Wang, Z. H.: PrePost+: A New Fast Vertical Method for Mining Frequent Itemsets. International Journal of Computational Intelligence Systems, Vol. 3, 2010, No. 6, pp. 733–744.

[14] Orlando, S.—Palmerini, P.—Perego, R.: Enhancing the Apriori Algorithm for Frequent Set Counting. In: Kambayashi, Y., Winiwarter, W., Arikawa, M. (Eds.): Data Warehousing and Knowledge Discovery (DaWaK 2001). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 2114, 2001, pp. 71–82.

[15] Orlando, S.—Palmerini, P.—Perego, R.—Silvestri, F.: Adaptive and Resource-Aware Mining of Frequent Sets. Proceedings of the ACM ICDM Conference on Data Mining, 2002, doi: 10.1109/ICDM.2002.1183921.

[16] Bayardo Jr., R. J.: Efficiently Mining Long Patterns from Databases. ACM SIGMOD Conference, 1998, doi: 10.1145/276304.276313.

[17] Agarwal, R.—Aggarwal, C. C.—Prasad, V. V. V.: Depth First Generation of Long Patterns. Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '00), 2000, pp. 108–118, doi: 10.1145/347090.347114.

[18] Agarwal, R.—Aggarwal, C. C.—Prasad, V. V. V.: A Tree Projection Algorithm for Generation of Frequent Itemsets. Journal of Parallel and Distributed Computing, Vol. 61, 2001, No. 3, pp. 350–371.

[19] Burdick, D.—Calimlim, M.—Gehrke, J.: MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases. Proceedings of the 17th International Conference on Data Engineering (ICDE), 2001, pp. 443–452, doi: 10.1109/ICDE.2001.914857.

[20] Lucchesse, C.—Orlando, S.—Perego, R.: DCI_Closed:: A Fast and Memory Efficient Algorithm to Mine Frequent Closed Itemsets. ICDM Workshop on Frequent Itemset Mining Implementations (FIMI '04), 2004.

[21] Lucchesse, C.—Orlando, S.—Perego, R.: Fast and Memory Efficient Mining of Frequent Closed Itemsets. IEEE Transactions on Knowledge and Data Engineering, Vol. 18, 2006, No. 1, pp. 21–36, doi: 10.1109/TKDE.2006.10.

[22] Pasquier, N.—Bastide, Y.—Taouil, R.—Lakhal, L.: Discovering Frequent Closed Itemsets for Association Rules. In: Beeri, C., Buneman, P. (Eds.): Database Theory – ICDT '99. Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 1540, 1999, pp. 398–416.

[23] Pasquier, N.—Bastide, Y.—Taouil, R.—Lakhal, L.: Efficient Mining of Association Rules Using Closed Itemset Lattices. Information Systems, Vol. 24, 1999, No. 1, pp. 25–46, doi: 10.1016/S0306-4379(99)00003-4.

[24] Pei, J.—Han, J.—Mao, R.: CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets. DMKD Workshop, 2000.

[25] Uno, T.—Kiyomi, M.—Arimura, H.: LCM Ver. 2: Efficient Mining Algorithms for Frequent/Closed/Maximal Itemsets. ICDM Workshop on Frequent Itemset Mining Implementations (FIMI '04), 2004.

[26] Wang, J.—Han, J.: BIDE: Efficient Mining of Frequent Closed Sequences. Proceedings of the 20th International Conference on Data Engineering (ICDE), 2004, doi: 10.1109/ICDE.2004.1319986.

[27] Wang, J.—Han, J.—Lu, Y.—Tzvetkov, P.: TFP: An Efficient Algorithm for Mining Top-k Frequent Closed Itemsets. IEEE Transactions on Knowledge and Data Engineering, Vol. 17, 2005, No. 5, pp. 652–664.

[28] Wang, J.—Han, J.—Pei, J.: CLOSET+: Searching for the Best Strategies for Mining Frequent Closed Itemsets. Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '03), 2003, pp. 236–245, doi: 10.1145/956750.956779.

[29] Zaki, M. J.—Hsiao, C.: ChARM: An Efficient Algorithm for Closed Association Rule Mining. SDM Conference, 2002.

[30] Extended BAKERY Dataset, `http://wiki.csc.calpoly.edu/datasets/wiki/ExtendedBakery`.

Savo Tomović received his Ph.D. in computer science from the University of Montenegro in 2011. He is currently Associated Professor in the Faculty of Science – Department of Mathematics and Computer Science at the University of Montenegro and Head of the Centre of the University Information System. He teaches a wide variety of undergraduate and graduate courses in several computer science disciplines, especially database systems, operating systems and programming. In addition, he is currently engaged as an adviser in Crnogorski Telekom on the project for data warehouse design and implementation. His primary research interest is in the area of data mining and artificial intelligence. During his Ph.D. studies he was involved in the project Linear Collider Flavour Identification (LCFI) with the aim to compare different data mining and classification algorithms as well as to understand the relative importance of the various input variables for the resulting tagging performance.

Predrag Stanišić is Full Professor in the Faculty of Science – Department of Mathematics and Computer Science at the University of Montenegro and Vice Chancellor of the University of Montenegro. He received his B.Sc. degree in mathematics and computer science from the University of Montenegro in 1996, his M.Sc. degree in computer science from the University of Belgrade, Serbia in 1998 and his Ph.D. degree in computer science from Moscow State University M. V. Lomonosov in 1999. He teaches a wide variety of undergraduate and graduate courses in several computer science disciplines, especially database systems, operating systems and programming.