Databases and Information Systems IX
G. Arnicans et al. (Eds.)

doi:10.3233/978-1-61499-714-6-211

The Choice of Optimal Algorithm for Frequent Itemset Mining

Vyacheslav BUSAROV, Natalia GRAFEEVA[1], Elena MIKHAILOVA
Saint Petersburg State University, St. Petersburg, Russia

Abstract. The algorithms for mining of frequent itemsets appeared in the early 1990s. This problem has an important practical application, so there have appeared a lot of new methods of finding frequent itemsets. The number of existing algorithms complicates choosing the optimal algorithm for a certain task and dataset. Twelve most widely used algorithms for mining of frequent itemsets are analyzed and compared in this article. The authors discuss the capabilities of each algorithm and the features of classes of algorithms. The results of empirical research demonstrate different behavior of classes of algorithms according to certain characteristics of datasets.

Keywords: data·mining, frequent·itemsets, average cover, transaction·database.

1. Introduction

The association rule mining task is to discover a set of attributes shared among a large number of objects in a given database. Consider, for example, the order database of a restaurant, where the objects represent guests and the attributes represent guests' orders. An example could be that "80% of people who order fish also order white wine". There are many other potential application areas for association rule technology, which include customer segmentation, catalog design, and so on.

The use of association rules for data analysis was first proposed in 1993 by R. Agrawal [1] who used them to analyze what is now referred to as market basket data. The idea behind association rules can be expressed as follows:

Let F be a set of items, and D a database of transactions, where each transaction has a unique identifier and contains a set of items. A set of items is also called an itemset. The support of an itemset f, denoted by $\sigma(f)$, is the number of transaction in which it occurs as a subset. An itemset f is frequent if its support is more than a user-defined minimum support value (MinSupport). Moreover, the value of support can be interpreted at absolute or relative format, it doesn't matter.

The association rule is an expression $A \Rightarrow B$, where A and B are non-overlapping itemsets. The support of rule is given as $\sigma(A \cup B)$, and the confidence as $\sigma(A \cup B)/\sigma(A)$, (i.e., the conditional probability that a transaction contains B, given that it contains A). The rule is confident if its confidence is more than a user-defined minimum confidence (MinConf).

The data mining task is to generate all association rules in the database, which have a support greater than minimum support, i.e., the rules are frequent. The rules

[1] Corresponding Author: n.grafeeva@spbu.ru

must also have confidence greater than minimum confidence, i.e. the rules are confident. The task can be broken into two steps [14]:

- Find all frequent itemsets. Given m items, there can be potentially 2^m frequent itemsets. Efficient methods are needed to traverse this exponential itemset search space to enumerate all the frequent itemsets. This frequent itemset discovery is the main focus of this paper.
- Generate confident rules. This step is relatively straightforward; rules of the form $A \backslash B \Longrightarrow B$, where $B \subset A$ are generated for all frequent itemsets A, provided the rules have at least minimum confidence.

We have already stated that finding frequently occurring datasets is an important subtask that helps answer a lot of questions. Since the variety of approaches is wide, practical applications require a reliable method of selecting proper algorithm to fit the task at hand. We look at the most widely used algorithms and some of the state-of-the-art approaches, namely:Apriori [2] 1994, Apriori Hybrid [3] 1994, FP-Growth[10] 2000, Eclat [14] 2000, dEclat [15] 2003, Relim[4] 2005, LCMFreq v.2/v.3[12,13] 2004-2005, H-mine[11] 2007, PPV[6] 2010, PrePost[7] 2012, FIN [8] 2014, PrePost+ [9] 2015. It is now our task to determine which of the algorithms above perform better than the others and should thus be chosen for optimal performance. This paper is a revised and extended version of [5].

2. Selection Criteria

Let us consider some of the possible criteria of optimality that we may use.

1. *Asymptotics*. In the theory of algorithms, this parameter is considered important, but it is completely devoid of objectivity in our case. The matter is that all algorithms use heuristics that allows reducing the search area. Their efficiency directly depends on the characteristics of particular data, its density and evenness of its distribution. In such cases, it is the worst case that is looked at, and the number of operations is determined on its basis, but such a situation occurs only while we are dealing with artificial and specially selected data that is not likely to be encountered in real life tasks. We will not use this criterion for comparing the algorithms.
2. *Simplicity of realization*. This criterion is undoubtedly quite subjective. The history of the IT industry offers telling examples of how this criterion was used as a basis for managerial decisions. In the 1990s, numerous attempts were made to pay remuneration to software engineers based on the complexity of their work. All such attempts, including those that involved counting the number of lines of code and measuring the time it took to write them, proved useless and unsuccessful. In an industry like the IT one, simplicity of realization cannot be a valid criterion because everything depends not only on a programming task itself, but also on a developer's programming background and implementation skills. In one of the reviewed works [4], however, the author uses simplicity of realization as one of the selection criteria for the purpose of his study. We consider this criterion to be intuitively clear, but we will not take it into consideration in comparing the algorithms.
3. *Volume of memory used.* This is a telling criterion. If measured in real experiments with the same data set, the volume of memory used can provide a reliable basis for

choosing the most effective algorithm for a particular case. It is important to note that, for the reasons described above, we will use an empiric evaluation rather than an asymptotic one. Thus, the volume of memory used will not be used as a criterion for evaluating the algorithms in our work.

4. *Execution time*. Since we have real life applications in mind, the empirically measured execution time of an algorithm will always be a criterion of utmost importance to us. It is this parameter that we will look at, first of all, while comparing the twelve most common algorithms.

3. Comparative Analysis of Algorithms

As was noted earlier, identifying frequently occurring itemsets is key to searching for association rules. Algorithms have to deal with large bases of initial data, which complicates analysis. It is, therefore, important how the incoming data stream is stored. All most widely used algorithms switch from attributes to symbols or sequences of symbols of a fixed length, thus limiting the task of storing initial data sets to storing a glossary of transactions. Therefore, the data structure chosen in this or that case is an important element of an algorithm.

The bulk of research devoted to discovering association rules focuses on two categories of algorithms [11]:

- "candidate-generation-and-test"
- "pattern-growth method"

A characteristic example of the first category of algorithms is Apriori (R. Agrawal, 1994 [2]). This category also includes all the subsequent variations of this algorithm based on the anti-monotony principle (the support of a set of items does not exceed the support of any of its subsets) [1]. Such algorithms generate itemsets of length $(k + 1)$ based on the previous itemsets having length k. Even though the anti-monotony property principle allows us to disregard quite a few variants, such algorithms are not efficient computationally if initial data is extensive (the number of itemsets or the length of sets).

A good example of the second category of algorithms is called FP-Growth (J. Han, 2000 [10]). Algorithms of this type perform in a recursive manner by breaking down a data set into several parts and looking for local results that are subsequently combined into an overall result. The algorithms of this type generate fewer candidates than the algorithms described above, which allows to save a considerable amount of memory. However, the productivity of such algorithms largely depends on the homogeneity of initial data.

The comparative analysis of the most widely used algorithms for finding frequent itemsets was based on studies published at different times. All the experiments described in those studies were run on well-known data sets in this subject area, such as: Pumsb, Mushroom, Connect, Chess and Accidents (FIMI repository – http://fimi.ua.ac.be). The Pumsb data set contains census data, while the Mushroom data set consists of characteristics of mushroom species, Connect и Chess are sources of data on progress in the corresponding games. The Accidents data contains information about traffic and accidents.

In order to obtain an overall view of the compared algorithms, we introduced binary directed relationships between the algorithms to reflect improvement of productivity and decreasing of memory used (based only on the experiments described in the articles mentioned above). According to the authors of the articles, each pair of algorithms was compared under identical conditions pertaining to the hardware characteristics, data, the language of realization, etc. Based on the relationships introduced between the algorithms, we made a chart of execution time (Figure 1) and a chart of memory requirements (Figure 2).

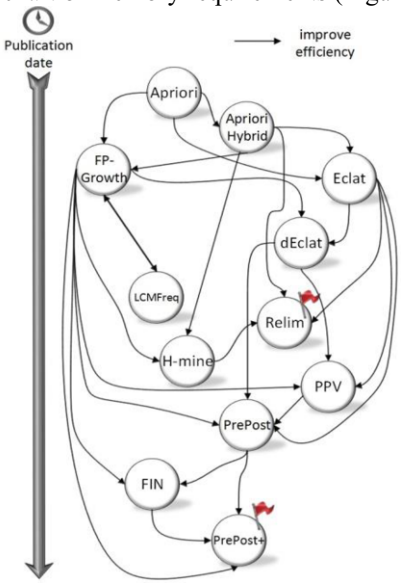

Fig. 1. The relationships between algorithms in terms of execution time [5].

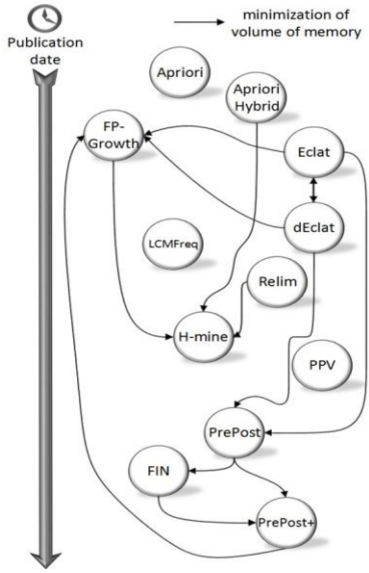

Fig. 2. The relationships between algorithms in terms of memory requirements [5].

The introduced binary directed relationships are transitive and, consequently, allow us to perform a comprehensive comparative analysis of all the algorithms. It should be noted that there is a two-direction relationship between some of the algorithms (for example, between FPGrowth and LCMFreq v.2/v.3in Figure 1, EClat and dEClat in Figure 2), which means that, depending on the type of initial data, such algorithms demonstrate approximately equal results and are considered on a par in this respect. Besides, while it is clearly seen in the first chart which algorithms outperform the others (they are marked with flags), it is impossible in principle to tell the "winner" in the second because there are algorithms that were not compared in terms of the volume of memory used (for example: Apriori, LCMFreqv.2/v.3, PPV) at articles, where they were described. Nevertheless, the second chart is useful in that it gives us some idea of the difference between some of the algorithms in terms of memory requirements.

Just like the authors of the studies we refer to, we consider execution time to be the most important characteristic. The experimental data shown on the charts above (Figure 1) leads to the conclusion that Relim and PrePost+ are the most efficient computationally. We will look at them in greater detail alongside with classical algorithms.

3.1. Apriori

This algorithm was proposed by the author of theory of association rules Rakesh Agrawal [2] in 1994 and is one of the first of its kind. As noted earlier, it is a typical representative of "candidate-generation-and-test" approach.

The main part of the algorithm (search of frequent itemsets) consists of several stages, each of which consists of the following steps:

- candidate generation
- candidate counting

Candidate generation is a step at which the algorithm scans the database and generates a lot of i-element of candidates where i is the number of stage. At this stage, the support of candidates is not calculated.

Candidate counting is a step at which the support is evaluated for each i-element candidate. Candidates whose support is less than the minimum set by the user (MinSupport) are also pruned at this step. The remaining i-element itemsets are considered frequent. The clipping of candidates is done on the basis of the anti-monotony property which states that all subsets of a frequent set must be frequent.

Of course, the anti-monotony property allows us not to consider all possible combinations of items, but even such a scan with clipping is computationally inefficient on large data sets.

3.2. FP-Growth

This algorithm was published in 2000 by J. Han [10] and was an important contribution to the field of mining frequent itemsets because it did not involve explicit generation of candidate sets. This approach was later called a pattern-growth method.

As was pointed out earlier, FP-Growth (Frequency-pattern-Growth) algorithm, just like many other algorithms, converts the task of looking for frequent sets to the problem of storing a relevant glossary, which problem is solved by using a prefix tree – an FP-Tree. At the beginning, items in each transaction are sorted in the order of descending support. Naturally, infrequent singleton items have already been discarded. The data structure itself is built in accordance with the following principles:

- The root of the tree$\mathrm{v}_0$is labeled as null.
- Each node v of the tree T consists: item $f_{\mathrm{v}} \in$F, a set of children nodes $S_{\mathrm{v}} \subseteq T$,
- support $\mathrm{c}_{\mathrm{v}} = v(\varphi_{\mathrm{v}}) : \varphi_{\mathrm{v}} = \{f_{\mathrm{u}} | \mathrm{u} \in [\mathrm{v}_0, \mathrm{v}]\}$, where $[\mathrm{v}_0, \mathrm{v}]$ is the path from the root of the tree v_0 to a node v.
- $V(T, f) = \{\mathrm{v} \in T | f_{\mathrm{v}} = f\}$ – all the nodes of the item f.
- $C(T, f) = \sum_{\mathrm{v} \in V(T,f)} c_{\mathrm{v}}$ –overall support of the item f.
- The tree is divided into levels, each of which corresponds to an item, and each item is associated with a single level. At the same time, the next node in the path can be at any level below the current one because all of them are sorted in the descending order of their support of corresponding items.

In the process of the algorithm, a conditional FP-tree (CFP-Tree) is repeatedly formed, such an FP-Tree being built only on transactions with a specified item. Let

there be an FP-tree T and item $f \in$F. Conditional FP-Tree $T' = T|f$ will be obtained if we:

1. Leave only the tree nodes on the path from the node v corresponding to the item f upwards to the root v_0: $T' := \bigcup_{v \in V(T,f)}[v, v_0]$.
2. Increase the value of support c_v of nodes $v \in V(T', f)$upwards according to the rule $c_u := \sum_{w \in S_u} c_w$ for each $u \in T'$.
3. Delete from T' all the nodes corresponding to the item f, because, by then, all the items lying below the item f will have already been considered.

It should be noted that T' is generated from the tree T without using the transaction database.

The final result is formed by means of a recursive procedure with the following parameters: FP-tree T, itemsetφ and a list of frequentitemsets R. As a result, all frequent itemsets containing φ, are added to the R. All the items $f \in$F: $V(T, f) \neq \emptyset$ are processed sequentially by levels of the tree from the bottom up, and if$C(T, f) \geq MinSupport$, then:

- $R := R \cup \{\varphi \cup \{f\}\}$
- A new tree is generated $T' = T|f$
- The procedure is started again with the parameters: $T', \varphi \cup \{f\}, R$.

Practice has shown that FP-Growth performs worse on test data than many other algorithms in terms of execution time (Figure 1), but the same experiments show that it makes a more efficient use of memory (Figure 2). This is attributed to what is a unique feature of an FP-Tree: if the relative density of the data you deal with is the same everywhere, then, beginning with a certain moment, adding new transactions to a tree will not cause the number of nodes to change.

3.3. Relim

This algorithm was proposed in 2005 in the study by Christian Borgelt [4]. The acronym "Relim" illustrates the underlying principle of the algorithm: "RecursiveELIMination scheme". Relim tries to find all frequent itemsets with a given prefix by lengthening it recursively and renewing support at the same time. The approach utilized here is called a "pattern-growth method". Let us look at the stages of the algorithm:

1. The first iteration on the transaction database allows to calculate support for each item separately and to exclude infrequent items from each transaction (in the example in Figure 4, MinSupport = 3). To improve the performance, the remaining items of each transaction are arranged in the order of ascending support.
2. In the next iteration, the array of lists is constructed as follows:
    - The head of each list is a particular frequent item. Lists are constructed for each of them.
    - Transactions starting with an item that corresponds to the head of the list are included in each list.

- Initially, the lengths of the corresponding lists are recorded in the cells of the array, but then the cells are used for storing relative supports of element headers in the context of a given prefix, that is $\{m \times v(\varphi) \mid \varphi \subseteq F$, prefix$\subseteq \varphi$, $v(\varphi)$ - support ofφ,m – the number of itemsets with this prefix$\}$. One of the main aspects of the algorithm is this: itemset support $(\varphi \cup f_i)$ is equal to the relative support f_i in the context of prefix φ.
- Then, the algorithms starts a recursive procedure with the following parameters: (1) the structure described above; (2) current prefix (initially empty), (3) minimal support.
- Next, the algorithm moves in two directions: a loop for a given data structure and a recursive sequence of procedure calls.
- In the loop, each element is dealt with separately, and a new data structure of the type described above is built on the basis of the list of transactions of each element header.
- The support is calculated for the itemset presented as the union $(\varphi \cup f_i)$ of the current prefix φ and the item f_i, and it is then added to the overall result if this itemset is a frequent one.
- The given element header is added to the current prefix and a recursive procedure with the resulting data structure and updated prefix is run.
- The lists of the original data structure and the structure obtained in the previous step are combined, the list of the current item header having been removed.
- Then, the next item in the loop is dealt with.

3.4. PrePost+

Proposed by Z. H. Deng [9] in 2015, it is the latest algorithm for identifying frequent itemsets. PrePost+ uses three data structures at a time: N-list, PPC-tree and set-enumeration tree, which explains why it requires more memory than FP-Growth does. Although it is an "Apriori-like" algorithm, it has empirically proved superior to many other algorithms in terms of execution time (Figure 1).

A PPC-tree is a tree structure:

1. It consists of one root labeled as "null", and a set of item prefix subtrees as the children of the root.
2. Each node in the item prefix subtree consists of five fields: item-name, count, children-list, pre-order, and post-order.

- item-name registers which item this node represents.
- count is the number of transactions presented by the portion of the path reaching this node.
- children-list contains all the children of the node.
- pre-order is the pre-order rank of the node.

- post-order is the post-order rank of the node.

For each node, its pre-order is the sequence number of the node when the tree is scanned by pre-order traversal, and its post-order is the sequence number when the tree is scanned by post-order traversal.

For each node N in a PPC-tree, we form <(N.pre-order,N.post-order):count> the PP-code of N.

The N-list of a frequent itemset is a sequence of all the PP-codes registering the item from the PPC-tree, with such PP-codes arranged in the ascending order of their pre-order values. The N-list of an itemset is formed, by using special rules, on the basis of the PP-codes corresponding to the items of the itemset.

A set-enumeration tree is a tree which consists of frequent items. All the items in the tree are arranged in the descending order of their support. Each node stores a single frequent item.

The framework of PrePost+ consists of the following:

- Constructing an PPC-tree and identify all frequent 1-itemsets;
- Constructing the N-list of each frequent 1-itemset on the basis of PPC-tree;
- Scanning the PPC-tree to find all frequent 2-itemsets;
- Mine all frequent k-itemsets (k>2).

The algorithm is based on two properties:

(1) For the given N-list of the itemset$\varphi \subseteq$F consisting of k items $\{< (x_1, y_1): z_1 >, < (x_2, y_2): z_2 >, \dots, < (x_k, y_k): z_k >\}$, we can calculate the support $v(\varphi) = \sum_{i=1}^{k} z_i$.

(2) $\forall \varphi \subseteq$ F $\forall f \in$ F : $v(\varphi) = v(\varphi \cup \{f\}) \Rightarrow \forall A \subseteq$ F : $A \cap \varphi = \emptyset, f \notin A$ the following is true:$v(\varphi \cup A \cup \{f\}) = v(\varphi \cup A)$.

Indeed, if $v(\varphi) = v(\varphi \cup \{f\})$, then any transaction containing φ, also contains f, from which the above identity obviously results from. The main difference between PrePost and PrePost+ lies in the strategy of pruning candidates for the status of frequent itemsets. PrePost+ uses itemset equivalence as a pruning strategy while PrePost utilizes the single path property of an N-list as a pruning strategy [7]. For the purpose of facilitating the mining process, PrePost+ uses a set-enumeration tree to provide the search space for frequent itemsets.

3.5. Review of Other Algorithms

Apriori Hybrid[3] (1994).*Category:* "candidate-generation-and-test".

Data structures used: a hash-tree or a hash-table is used for storing generated candidate sets (it simplifies the calculation of support for new sets), and two-dimension number array for storing frequent itemsets. Each of such itemset has a unique identifier. It is these identifiers that are used for indexing.

Some key features: the algorithm applies the same principles as Apriori does, but it does not refer to the initial transaction database in order to calculate the support of each itemset. Instead, the techniques of hashing and intersecting the itemsets are used.

Eclat [14] (2000).*Category:* "candidate-generation-and-test".

Data structures used: what is called Lettuces is partially ordered sets, in which each pair of elements has unique supremum and infinum.

Some key features: all the candidates are stored in a special data structure called Lattuces. During the search, the algorithm passes this data structure widthwise and

depthwise. One of the main heuristics is that the algorithms strives to partition the itemset into subsets and look at them separately. In the process of looking for frequent itemsets, Eclat tries to identify the equivalence classes, thus pruning the range of possible candidates.

dEclat [15] (2003).*Category:* "candidate-generation-and-test".

*Data structures used:*diffset is a data structure that is capable of storing, intersecting and combining itemsets.

*Some key features:*dEclat is a modification of the previous algorithm by the same authors [14]. The main departure of this algorithm from Eclat is the use of a new data structure that allows to prune a large number of candidates. This algorithm also uses the concept of equivalency based on the values of the function defined for itemsets. One diffset stores equivalence classes, and the other diffset stores the prefixes of candidates of varying length.

LCMFreqv.2/v.3 [12][13] (2004-2005).*Category:* "pattern-growth method".

Data structures used: this algorithm uses a combination of common structures – bitmap, prefix tree and array list.

Some key features: a prefix tree contains possible candidates for frequent itemsets, the order of items being clearly fixed. Original transactions are stored in an array list. Bitmap data structure is used for calculating support in a more efficient way. The shared use of these data structures varies from one version of the algorithm to another. V.3 is considered by the authors of the algorithm to be the most efficient of all. It is this version that we used for our experiments. Further in the text, we refer to it as LCMFreq.

H-mine [11] (2007).*Category:* "pattern-growth method".

Data structures used: H-struct is a data structure that stores frequent itemsets together with references to the corresponding transactions in the original transaction database. The method of its construction is similar to that of FP-Growth; however, instead of storing transactions explicitly, the algorithm operates with references to them. It is thanks to this feature that H-mine outperforms many algorithms in the efficiency of memory use (as you can see in Figure 2).

Some key features: the method of scanning H-struct is basically the same as that in FP-Growth. What differentiates one of the two algorithms from the other is the approach to storing initial data.

PPV[6] (2010).*Category:* "candidate-generation-and-test".

Data structures used: a PPC-tree is the same prefix tree as the one used in the algorithm PrePost+. Node-list is a data structure that consists of PP-codes formed on the basis of a PPC-tree. It is absolutely identical to N-list, the only difference between Node-list and N-list being that Node-list use descendant nodes to represent an itemset while N-list represent an itemset by ancestor nodes.

Some key features: the PPC-tree stores the original transactions. Candidate for the status of frequent itemsets are stored in the Node lists. The problem of calculating the support of candidates boils down to the operation of intersecting Node-lists, which is done in linear time. The operation of intersecting Node-lists is the key feature of PPV algorithm.

PrePost [7] (2012).*Category:* "candidate-generation-and-test".

*Data structures used:*a PPC-tree is the same prefix tree as the one utilized in PrePost+. The other data structure used in this algorithm is N-list that consists of PP-codes formed on the basis of a PPC-tree.

Some key features: as noted by the authors themselves, the main difference between PrePost and PrePost+ is the pruning strategies used: PrePost+ adopts itemset equivalence as the pruning strategy whereas PrePost utitizes single path property of N-list for this purpose. To facilitate the mining process, PrePost+ uses a set-enumeration tree to represent the search space for frequent itemsets. The remaining steps of these algorithms are similar to each other.

FIN [8] (2014).*Category:* "candidate-generation-and-test".

*Data structures used:*a Node-set is a structure that is identical to N-list and Node-list, but it uses twice as little memory since it requires to store either pre-order parameter or post-order one instead of storing both of them at a time. A POC-tree (Pre-Order Coding tree) is a prefix tree that is absolutely identical to a PPC-tree, but it does not store parameter post-order at all. A set-enumeration tree is a tree that stores all possible items in the ascending order of their support. Each node of the tree stores a single frequent item.

Some key features: in many aspects, this algorithm is similar to Pre-Post, the used data structures being the only difference between the two algorithms.

4. Experiments

The chart of relationship shown in Figure 2 is meaningful because it is built on the basis of real empirical evidence. The facts that each pair of algorithms was compared on the same databases (1), that the algorithms in each pair were realized in one and the same programming language (2), and that the same computing equipment was utilized in comparing each pair of algorithms (3), provide a valid basis for concluding which of the twelve algorithms is the most efficient. However, the fact that the chart of relationship was built based on separate comparisons is a drawback of this chart as a whole. It is for this reason that we ran our own experiments and compared the performance of all the twelve algorithms in likely conditions in terms of execution time.

For the purposes of this study, we selected the Mushroom data set mentioned almost in all the reviewed publications. The average transaction length is 23, the glossary of attributes consists of 119 elements, and the total number of transactions amounts to 8124. The data contain a description of mushroom species and their characteristics. We used implementations, which exactly corresponded author's algorithms and pseudocodes. All the experiments were run on a computer with the following processor: Intel(R) Core(TM) i7-4700HQ CPU @ 2.40GHz, 12.0 GB RAM. All the algorithms were realized on Java 8.

Table 1. Execution time of Mushroom experiments (sec). *Source:[5].*

Minimal support	Pre Post+	Relim	FIN	PrePost	PPV	H-mine	dEclat	FP-Growth	LCM Freq	Eclat	Apriori Hybrid	Apriori
70%	401,1	397,6	412,5	415,2	417,6	419,4	426,1	427,9	429,9	434,9	437,5	438,2
75%	373,3	372,2	383,8	385,1	389,9	393,0	403,4	404,1	409,1	416,1	419,2	422,9

80%	351,8	341,1	362,9	365,1	373,3	374,89	379,8	377,0	382,6	391,2	392,9	395,0
85%	337,6	332,9	349,5	348,6	358,8	360,2	373,1	375,5	380,3	390,7	394,1	394,1
90%	328,1	327,9	340,1	345,6	349,1	350,4	359,3	362,9	364,8	375,2	378,9	379,0

The value of minimal support largely affects the number of sets in the result, their length and, of course, execution time. For greater clarity, we measured the value of minimum support (MinSupport) by percentage, with this value defined as the ratio of the number of transactions containing such itemsets to the total number of transactions. As follows from the definition above, this value cannot exceed 100%. By varying this value, we finally arrived at the results presented in Table 1.

These results are graphically represented in Figure 3.

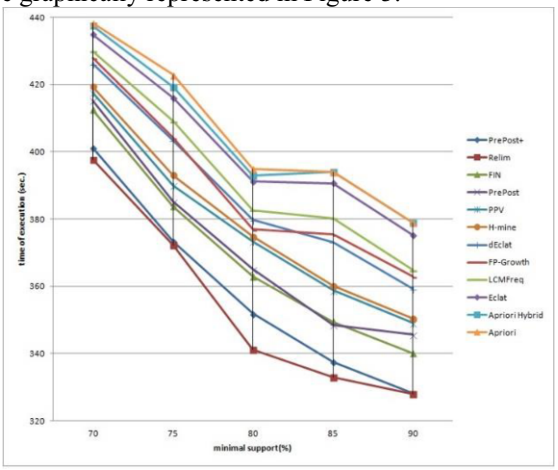

Fig. 3. The chart of execution time of Mushroom experiments (sec).

As shown by the experiments, the algorithms for finding frequent item sets rank as follows in terms of execution time: Relim (1), PrePost+ (2), FIN (3), PrePost (4), PPV (5), H-mine (6), dEclat (7), FP-Growth (8), LCMFreq (9), Eclat (10), Apriori Hybrid (11), Apriori (12). What is most important is that the results of our experiments are fully consistent with the relationship chart of execution time which was built on the basis of previous studies of the algorithms.

But many authors declare that the execution time of the algorithms depends heavily on the specifics of the data. So we selected another well-known Chess dataset mentioned in many reviewed publications for further research. The dataset contains the results of chess games. The average transaction length is 6, the glossary of attributes consists of 36 elements, and the total number of transactions amounts to 28056. We have conducted experiments with this dataset and got the results shown in Table 2.

Table 2. Execution time of Chess experiments (sec).

Minimal support	Pre Post+	Relim	FIN	PrePost	PPV	H-mine	dEclat	FP-Growth	LCM Freq	Eclat	Apriori Hybrid	Apriori
70%	807,2	788,2	825,8	838,8	833,6	840,8	853,2	854,2	858,4	880,6	874,4	883,4
75%	740,6	735,8	753,9	778,9	773,2	787,8	804,4	816,8	818,1	832,1	838,5	845,8
80%	703,7	682,2	723,8	742,2	736,6	749,6	759,6	762,0	765,2	782,4	785,7	790,0
85%	678,1	675,4	698,9	709,1	705,5	720,5	746,3	751,0	760,3	780,5	789,0	789,1
90%	648,2	647,9	672,3	672,2	690,2	700,8	710,5	716,1	724,9	740,5	751,9	754,0

These results are graphically represented in Figure 4.

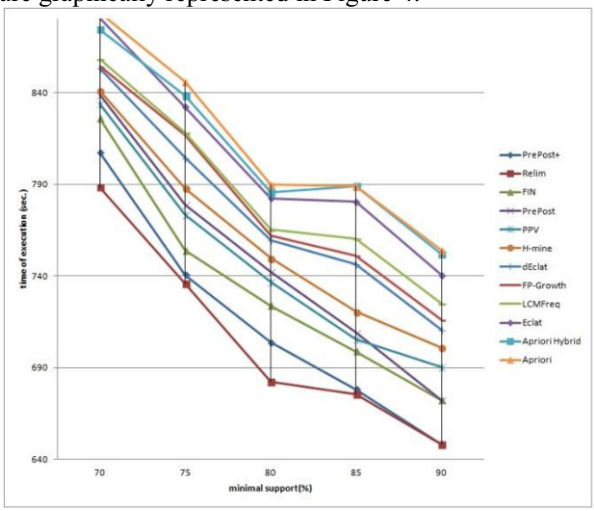

Fig. 4. The chart of execution time of Chess experiments (sec).

We see that there are no fundamental differences in the relative efficiency of algorithms. The rating of the algorithms in terms of execution time has not changed. What is common in data structures of the selected datasets? It turns out that they are similar in the terms of indicator, which is called “average cover of a glossary". This concept for a dataset is defined as follows:

$$average\ cover\ of\ a\ glossary = \frac{1}{n}\sum_{i=1}^{n}\frac{|\tau_i|}{|D|} * 100$$

where$D$ - the glossary of attributes, τ_i – transactions, n - the number of transactions.

Datasets Mushrooms and Chess have average cover indicators equal to 23.7% and 22.1%, respectively. However other datasets can have very different values of average cover indicator. How will the algorithms work with such datasets? Since public datasets do not have sufficient flexibility to more thoroughly examine this question, we wrote a utility to generate random datasets according to the specified parameters: the number of transactions, the capacity of a glossary and the average cover of a glossary.

We have generated a glossary consisted of 70 items, datasets consisted of 90 000 transactions, and varied the value of the average cover of the glossary. We have conducted experiments with a minimum support (MinSupport) equal to 80 and got the results shown in Table 3.

Table 3. Execution time of experiments with different values of average cover (sec).

Avg cover	Pre Post+	Relim	FIN	PrePost	PPV	H-mine	dEclat	FP-Growth	LCM Freq	Eclat	Apriori Hybrid	Apriori
2%	2104	2356	2197	2188	2191	2446	2211	2516	2552	2381	2390	2461
5%	2168	2294	2231	2215	2206	2374	2226	2445	2471	2386	2414	2493
10%	2237	2263	2280	2261	2265	2306	2269	2405	2423	2419	2452	2551
15%	2255	2192	2270	2249	2255	2237	2275	2310	2350	2440	2462	2574
25%	2270	2208	2299	2312	2324	2340	2345	2348	2363	2478	2491	2590

It is interesting to note that the results of the experiments in Table 3 clearly demonstrate the difference in behavior of "candidate-generation-and-test" and "pattern-growth method" algorithms. If we place the results on different charts this difference becomes more noticeable (Figure 5 and Figure 6).

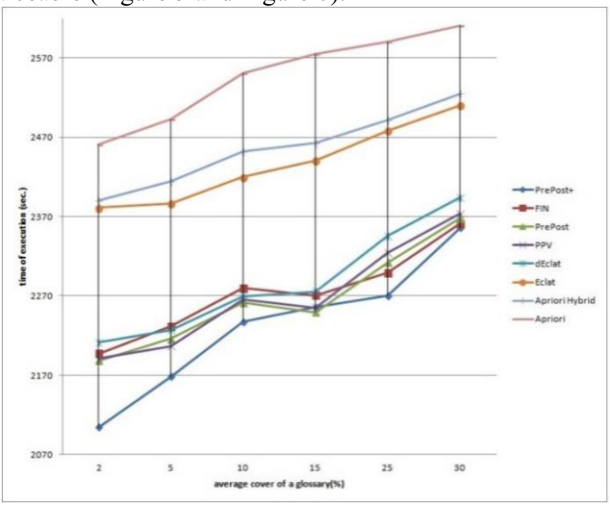

Fig. 5. The chart of execution time of "candidate-generation-and-test" algorithms (sec).

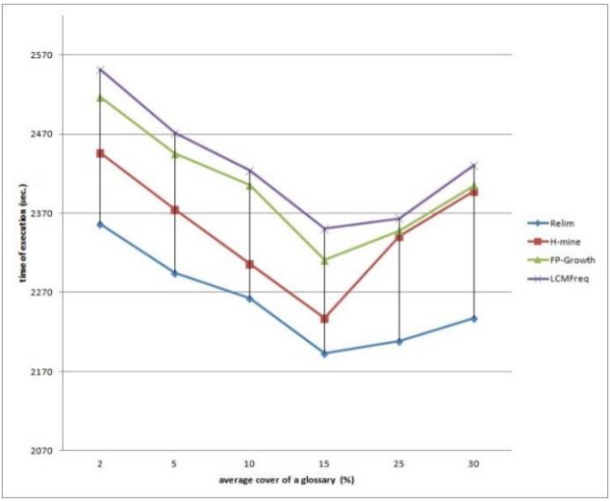

Fig. 6. The chart of execution time of "pattern-growth method" algorithms (sec).

Certain patterns of behavior common to all algorithms of each type become visible on these charts. "Candidate-generation-and-test" algorithms are better for small values of the average cover of a glossary (2-15%). At the same time, they are noticeably less efficient for large values of this index (15-30%). This fact may be explained: for small values of the average cover of a glossary heuristics effectively cut off unsuitable candidates, that improves the speed of the algorithm work. The growth of the average cover of a glossary makes heuristics work more and more rarely. It finally decreases the efficiency of the whole algorithm. However in this interval "pattern-growth method" algorithms effectively use their main advantage – the search of local results with their further extension.

5. Conclusion and Ideas for Further Research

We hope that the results of our research can be useful for developers for choosing the appropriate algorithm for mining of frequent itemsets according to the characteristics of the dataset. The simple calculation of the average cover of a glossary in the selected dataset allows at least to determine the appropriate type of an algorithm ("candidate-generation-and-test" or "pattern-growth method") and to restrict the search of the most suited algorithm. We are planning to expand this study to cover any new approaches to finding frequent itemsets if such approaches appear. It is also our intention to undertake a more serious comparison of the algorithms both in terms of execution time and memory requirements by running experiments on large databases, such as Kosarak, the latter database notably containing 990002 transactions and 41270 possible attributes. It also makes sense in the future to look at the implementation of these algorithms in distributed environments.

References

[1] Agrawal R., Imielinski T., Swami A.:Mining associations between sets of items in large databases. In: ACM SIGMOD Int. Conf. on Management of Data (SIGMOD 1993,Washington, D.C), 207-216 (1993)
[2] Agrawal R., Srikant R.: Fast algorithms for mining association rules. In: 20th Int. Conf. on Very Large Databases (VLDB 1994, Santiago, Chile), 487-499(1994)
[3] Agrawal R., Mannila H., Srikant R., Toivonen H.: Fast discovery of association rules. Advances in knowledge discovery and data mining. AAAI MIT Press, 12(1), 307-328 (1996)
[4] Borgelt C.: Keeping things simple: finding frequent item sets by recursive elimination. In: Open Source Data Mining Workshop (OSDM'05, Chicago, IL), 66-70. ACM Press, New York (2005)
[5] Busarov V., Grafeeva N., and Mikhailova E.: A Comparative Analysis of Algorithms for Mining Frequent Itemsets, Proceedings of the 12th International Baltic Conference, DB&IS 2016, pp. 136–150, Springer (2016)
[6] Deng Z. H., Wang Z.: A New Fast Vertical Method for Mining Frequent Patterns. International Journal of Computational Intelligence Systems, 3(6), 733-744 (2010)
[7] Deng Z. H., Wang Z., Jiang J.: A new algorithm for fast mining frequent itemsets using N-Lists. Science China Information Sciences, 55 (9), 2008-2030 (2012)
[8] Deng Z. H., Lv S. L.: Fast mining frequent itemsets using Nodesets. Expert Systems with Applications, 41(10), 4505–4512 (2014)
[9] Deng Z. H., Lv S. L.: PrePost+ : An efficient N-lists-based algorithm for mining frequent itemsets via Children–Parent Equivalence pruning. Expert Systems with Applications, 42(10), 5424 – 5432 (2015)
[10] Han J., Pei H., Yin Y.: Mining frequent patterns without candidate generation. In: ACM SIGMOD Int. Conf. on Management of data (SIGMOD 2000, Dallas, TX), 1-12 (2000)
[11] Pei J., Han J., Lu H., Nishio S., Tang S., Yang D.: H-Mine: Fast and space-preserving frequent pattern mining in large databases. IIE Transactions, vol. 39(6), 593–605 (2007)
[12] Uno T., Kiyomi M., Arimura H.: LCM ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In: Workshop on Frequent Itemset Mining Implementations (FIMI'04, Brighton, UK) (2004)
[13] Uno T., Kiyomi M., Arimura H.: LCM ver.3: Collaboration of array, bitmap and prefix tree for frequent itemset mining. In: Open Source Data Mining Workshop (OSDM'05, Chicago, IL), 77-86. ACM Press, New York (2005)
[14] Zaki M. J.: Scalable Algorithms for association mining. IEEE transaction on knowledge and data engineering, vol.12, no. 3, 372-390 (2000)
[15] Zaki M. J., Gouda K.: Fast vertical mining using diffsets. In: 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD 2003, Washington, DC), 326–335. ACM Press, New York (2003)