# X-FSPMiner: A Novel Algorithm for Frequent Similar Pattern Mining

ANSEL Y. RODRÍGUEZ-GONZÁLEZ, Unidad de Transferencia Tecnológica, Centro de Investigación Científica y de Educación Superior de Ensenada, Tepic, México
RAMÓN ARANDA, Centro de Investigación en Matemáticas, A.C., Unidad Mérida, Mérida, México
MIGUEL Á. ÁLVAREZ-CARMONA, Centro de Investigación en Matemáticas, A.C., Unidad Monterrey, Apodaca, México
ANGEL DÍAZ-PACHECO, Universidad de Guanajuato, División de Ingenierías, Campus Irapuato-Salamanca, Salamanca, México
ROSA MARÍA VALDOVINOS ROSAS, Facultad de Ingeniería, Universidad Autónoma del Estado de México, Toluca de Lerdo, México

Frequent similar pattern mining (FSP mining) allows found frequent patterns hidden from the classical approach. However, the use of similarity functions implies more computational effort, becoming necessary to develop more efficient algorithms for FSP mining. This work aims to improve the efficiency of mining all FSPs when using Boolean and non-increasing monotonic similarity functions. A data structure to condense an object description collection named *FV-Tree*, and an algorithm for mine all FSP from the *FV-Tree*, named *X-FSPMiner*, are proposed. The experimental results reveal that the novel algorithm *X-FSPMiner* vastly outperforms the state-of-the-art algorithms for mine all FSP using Boolean and non-increasing monotonic similarity functions.

CCS Concepts: • **Information systems → Data mining**; • **Theory of computation → Sorting and searching**.

Additional Key Words and Phrases: data mining, frequent patterns, similarity functions, mixed data



## 1 INTRODUCTION

In the last decade of the previous century, frequent pattern mining [14] emerged from the market basket analysis, playing an essential role in other data mining tasks like association rule mining, classification, clustering and prediction [1, 8, 34]. Frequent itemsets were discovered in transactional data, in which each transaction is the set of items purchased by a customer [2]. A frequent itemset is a set of items that occurs at least in the minimum number of transactions. To prune the search space of frequent itemsets, a property named downward closure property (i.e., all supersets of a non-frequent itemset are non-frequent itemsets) is used.

In collections of more complex objects described by numerical and not numerical features (e.g., electronic medical records [19], crime databases [40], sociological databases [23] and educational data [43]) also frequent

Authors' addresses: Ansel Y. Rodríguez-González, ansel@cicese.edu.mx, Unidad de Transferencia Tecnológica, Centro de Investigación Científica y de Educación Superior de Ensenada, Tepic, Andador 10, 109, Nayarit, México, 63173; Ramón Aranda, arac@cimat.mx, Centro de Investigación en Matemáticas, A.C., Unidad Mérida, Mérida, PCTY, Yucatán, México, 97302; Miguel Á. Álvarez-Carmona, miguel.alvarez@cimat.mx, Centro de Investigación en Matemáticas, A.C., Unidad Monterrey, Apodaca, Alianza Centro 502, Nuevo León, México, 66629; Angel Díaz-Pacheco, angel.diaz@ugto.mx, Universidad de Guanajuato, División de Ingenierías, Campus Irapuato-Salamanca, Salamanca, Carretera Salamanca - Valle de Santiago km 3.5 + 1.8 , Guanajuato, México, 36885; Rosa María Valdovinos Rosas, rvaldovinosr@uaemex.mx, Facultad de Ingeniería, Universidad Autónoma del Estado de México, Toluca de Lerdo, Instituto Literario 100, México, 50000.


ACM 1556-4681/2024/1-ART
https://doi.org/10.1145/3643820

patterns were mined in the subsequent years. However, commonly in a data preprocessing step, new features are created from the existing ones, conversing each numerical feature into several nominal features. Then an itemset mining algorithm is applied.

While the classical approach of frequent pattern mining counts object sub-descriptions using exact matching, other similarity functions [7, 35, 36, 38] to compare objects are widespread in soft and hard sciences. For example, in geology [15], medicine [4, 22] and sociology [31], two instances or objects can be considered similar, even if they are not identical. In these problems, the similarity is used to compare object sub-descriptions, count how many times an object sub-description appears in an object collection, and make decisions. From this fact, [9] proposed complex mining objects (described by numerical and not numerical features) using comparison criteria for each feature and a similarity function over the object sub-descriptions. Later [26] formalizes these preliminary ideas, extending the concept of downward closure property to the use of similarity functions, linking it to properties of similarity functions, and defining the frequent similar pattern mining (FSP mining) problem.

A frequent similar pattern is a combination of feature values of the study objects, such that the similarity accumulation of its similar patterns is not less than a user-specified frequency threshold. The FSP mining problem discovers all FSP from an object description collection, given a user-specified frequency threshold and a similarity function.

The frequent similar pattern mining approach allows found frequent patterns hidden for the classical approach. Several algorithms have been proposed to mine FSPs: *ObjectMiner* [9], *STreeDC-Miner* [26, 29], *STreeNDC-Miner* [26, 29], *RP-Miner* [28], *CFSP-Miner* [25], *STree*$^*$*DC-Miner* [27, 30], *STree*$^*$*NDC-Miner* [30] and *RP*$^*$*-Miner* [30]. Also, [29] shown that in tasks like classification, the accuracy using the FSP is higher than the accuracy using the frequent patterns obtained by the classical approach.

Developing more efficient algorithms is a research issue for both the classical and FSP mining approaches. However, the use of similarity functions by the FSP mining approach carries more computational effort, which requires additional attention.

This paper focuses on FSP mining using Boolean and non-increasing monotonic similarity functions. The main contributions are i) a novel data structure, named *FV-Tree*, to condense an object description collection; ii) a novel algorithm for mine all FSP that use *FV-Tree*, named *X-FSPMiner*. The experimental results show that the proposed algorithm (*X-FSPMiner*) is faster than the state-of-the-art algorithms for mine all FSP using Boolean and non-increasing monotonic similarity functions.

The outline of this paper is as follows. In Section 2 related work is reviewed. Section 3 provides the basic concepts and notation of FSP mining. In Section 4 the *FV-Tree* data structure and the *X-FSPMiner* algorithm for mining frequent similar patterns is proposed. Section 5 presents the experimental results and discussion, and finally, in Section 6 some conclusions and future work are discussed.

## 2 RELATED WORK

Two related approaches for frequent pattern mining can be distinguished in the literature: the classical frequent pattern mining, which uses exact matching, and the frequent pattern mining, based on the similarity. In this section, the main studies about them are included.

### 2.1 Frequent Pattern Mining

The first work-related to frequent mining itemset was proposed by Agrawal et al. [2]. After that work, many methods have been proposed, and in general, those methods can be mainly classified into two groups: *Apriori*–like and *Frequency Pattern growth* (*FP–growth*) like methods [6]. *Apriori*–like methods are based on the anti–monotony principle [20]. Thus, these methods generate candidate itemsets of length $p+1$ from frequent itemsets of length

$p$ by scanning the database iteratively. [3, 32, 33, 41, 42]. Some disadvantages of these methods are that they need to scan the database many times and generate a large set of candidates [17].

Differently to *Apriori*–like methods, *FP–growth*–like methods, instead of generating candidate itemsets, use strategies to recursively search frequent local patterns by dividing the dataset into sub-datasets. Then, the frequent local patterns are assembled into more extended global frequent patterns. Thus, *FP–growth*–like methods reduce search space and generate frequent itemsets without candidate generation. Some of those methods are based on *FP-Tree* to encode the dataset, and then they extract the frequent itemsets from the tree [16, 18, 21, 24]. A *FP-Tree* is represented by a *frequent–item header table* and *prefix subtrees* where each node consists of three fields: *item–name* (item this node represents), *count* (number of transactions obtained in the path portion reaching this node), and the *node–link* (links to the next node in the *FP-Tree* with the same item–name). *FP-Tree* has shown to be a highly condensed data structure for storing the database. However, it has been observed that the generation and use of the *FP-Trees* can be complex and inefficient in sparse dataset [39].

To overcome mentioned disadvantages, other *FP–growth*–like methods adopt a prefix tree structure called Pre–Post Code tree (*PPC-Tree*) to store the dataset [5, 10–13, 37]. Each node in the *PPC-Tree* includes 5 fields: *item–name* (item this node represents), *count* (number of transactions obtained in the portion of the path reaching this node), *children–list* (it registers all children of the node), *pre–order* (pre–order traversal code), and *post–order* (post–order traversal code). Note that although *FP-Tree* and *PPC-Tree* have similar structures, their main difference is that *PPC-Tree* does not handle a *header table*, which makes it simpler than *FP-Tree*.

Inspired by the *FP-Tree* and the *PPC-Tree*, one of this work's main contributions is the proposal of a novel data structure for frequent similar pattern mining, named *FV-Tree*. This novel structure is described in detail in section 4.

## 2.2 Frequent Similar Pattern Mining

In the literature, there are several algorithms for mining FSP, which can be classified in a two-dimensional space taking into account the image and the monotony of the similarity functions allowed (See figure 1).

The image of similarity functions can be Boolean (in $\{0, 1\}$, 1 means that objects compared are similar and 0 means the opposed) or non-Boolean (in $[0, 1]$, where the closer the similarity is to 1, the more similar are the objects compared). The monotony of similarity functions can be non increasing or increasing. A similarity function is non-increasing monotonic if and only if, for any pair of an object, the similarity regarding a set of features is greater than or equal to the similarity regarding any superset of features. The non-increasing monotony of a similarity function is relevant property because it implies that all super-descriptions of a non-FSP are also non FSPs. This property, known as $f_S$-downward closure property, allows pruning the search space of FSPs [29, 30].

This work aims to improve the efficiency of mining all FSPs using Boolean and non-increasing monotonic similarity functions. The state-of-the-art algorithms for this kind of similarity function are *ObjectMiner*, *STreeDC-Miner* and *CFSP-Miner*. However, *CFSP-Miner* does not mine all the FSPs, only a subset of them. Therefore, in the rest of the work, we only focus on *ObjectMiner* and *STreeDC-Miner*.

*ObjectMiner* [9] was the first algorithm for FSP mining. It was inspired on the Apriori algorithm [3]. The algorithm starts identifying all FSPs with only one feature, next following a breadth-first search strategy, for each iteration, $k$ (starting with $k = 2$), a set of candidates to FSPs with $k$ features is obtained. For that, pairs of FSPs with $k - 1$ features are merged to obtain FSPs with $k - 2$ features values exactly equal. For each $P$ candidate to FSP obtained by merging a pair $(P_1, P_2)$ of FSP, a set of candidate objects that contains a sub-description similar to $P$ is obtained. The frequency of each candidate to FSP is computed, iterating only over its set of candidate objects to contain similar sub-descriptions and using the similarity function. If the frequency is greater than the minimum frequency threshold, then the candidate to FSP is an FSP. *ObjectMiner* finishes when an iteration $k$

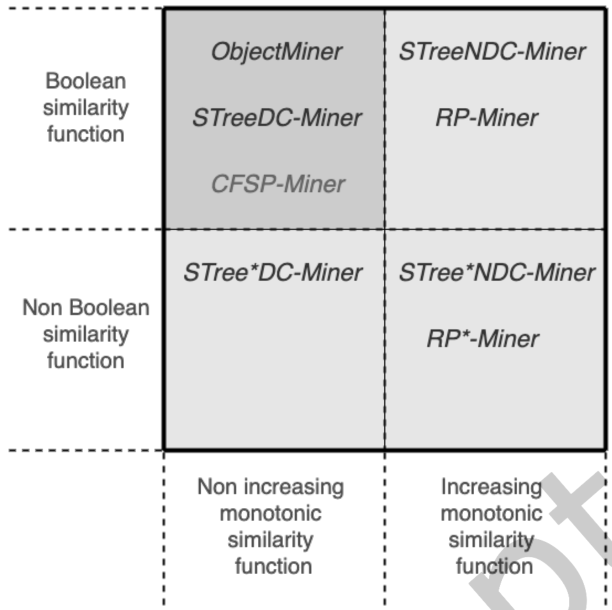


Fig. 1. Frequent similar pattern mining algorithms



does not produce any FSP with $k$ features. The main weakness of *ObjectMiner* is their high computational cost for the FSP mining process and the storage cost of the repetitions of the FSP sub-descriptions.

*STreeDC-Miner* [29] solved the main weakness of *ObjectMiner* storing the sub-descriptions of objects into groups of equals sub-descriptions. Consequently, never two equals sub-description are compared, and never repetitions of different sub-descriptions are compared. To do this, *STreeDC-Miner* introduces a tree structure called *STree*. For each set of features $A$, each leaf of the associated tree structure $STree_A$ represents a sub-description concerning the set of features $A$ and stores all its repetitions and the similarities with other sub-descriptions (other leaves). The branches of the $STree_A$ contain the common prefixes of the stored sub-descriptions. *STreeDC-Miner* sets an explicit definition of a total order over the feature set. From each set of only one feature $A$, and following a depth-first search strategy, a recursive procedure adds to $A$ in each call, a new feature greater than the features in $A$. Also, in each call, a tree structure $STree_A$ is built, the frequency of the sub-descriptions in $STree_A$ is computed, and the FSPs are obtained. If the current set of features $A$ contains only one feature, $STree_A$ is built from the dataset. Otherwise, $STree_A$ is built from the tree structure built in the previous recursive call. The recursive procedure's base case is produced when there are no frequent similar patterns for the set of feature $A$ or no feature to add. In the *STree* building process, the similarity between two sub-descriptions is only computed if their sub-descriptions in the tree structure built in the previous recursive call are similar, and almost one of them is an FSP. Consequently, the number of similarity function evaluations is reduced, and the computational effort to compute the frequency of each sub-description is reduced. Thus, the behavior of the *STreeDC-Miner* surpassed the behavior of the *ObjectMiner*. However, the successive creation (and destruction) of tree structures, one tree structure for each feature set, is time-consuming.

To address the weakness of the *STreeDC-Miner*, in this work, we propose a single tree structure to condense the dataset and a novel FSP mining algorithm that uses it.

## 3 BASIC CONCEPTS AND NOTATION

This section provides the basic concepts and notation of FSP mining used in the rest of the work. We follow the notations and definitions reported in [29].

Let $\Omega = \{O_1, O_2, \ldots, O_n\}$ be an object collection. Each object $O_i$ is described by a set of feature $R = \{r_1, r_2, \ldots, r_m\}$ and represented as a tuple $(v_1, v_2, \ldots, v_m)$, where $v_j \in D_j$ ($D_j$ is the domain of the feature $r_j$, $1 \leq j \leq m$). A *subdescription* of an object $O$ for a subset of features $S \subseteq R$ denoted by $I_S(O)$, is the description of $O$ in terms of the features in $S$; $O[r]$ denotes the value of the feature $r \in R$ of $O$.

Also, $f_S : \Omega \times \Omega \longrightarrow \{0, 1\}$ is a Boolean similarity function to compare two object descriptions regarding a set o features $S$. Given two subdescriptions $I_S(O)$, $I_S(O')$, with $O, O' \in \Omega$, $f_S(O, O') = 1$ means that $O$ is similar to $O'$ with respect to $S$ and $f_S(O, O') = 0$ means that $O$ is not similar to $O'$ with respect to $S$. Two examples of Boolean similarity functions [26, 28, 29] used in the FSP mining approach are:

$$f_S(O, O') = \begin{cases} 1 \text{ if } \forall r \in S, C_r(O[r], O'[r]) = 1 \\ 0 \text{ otherwise} \end{cases} \tag{1}$$

$$f_S(O, O') = \begin{cases} 1 \text{ if } \frac{|\{r \in S | C_r(O[r], O'[r]) = 1\}|}{|S|} \geq k \\ 0 \text{ otherwise} \end{cases} \tag{2}$$

where $C_r : D_r \times D_r \longrightarrow \{0, 1\}$ is a comparison function between values of feature $r$, and $k \in [0, 1]$. Although other comparison criteria can be used, the next two are the most commonly used on related works:

$$C_r(x, y) = \begin{cases} 1 \text{ if } x = y \\ 0 \text{ otherwise} \end{cases} \tag{3}$$

$$C_r(x, y) = \begin{cases} 1 \text{ if } |x - y| \leq \varepsilon \\ 0 \text{ otherwise} \end{cases} \tag{4}$$

Equation (3) compares non numerical features values and equation (4) compares numerical features values.

Now, the *frequency* of $I_S(O)$ in $\Omega$ for $f_S$ is defined as:

$$freq_{f_S,\Omega}(O) = \frac{|\{O' \in \Omega \ : f_S(O, O') = 1\}|}{|\Omega|}. \tag{5}$$

Using this frequency definition, $I_S(O)$ is a *$f_S$-frequent subdescription* (a frequent similar pattern) in $\Omega$ if $freq_{f_S,\Omega}(O) \geq minFreq$, where $minFreq$ is a given minimum frequency threshold. Consequently, the frequent similar pattern mining problem consists in finding all frequent similar patterns in $\Omega$.

Analogously to the frequent itemset mining, where downward closure property was defined and used to prune the search space, for similar frequent pattern mining, a downward class property was also defined [29]:

PROPERTY 1 ($f_S$-DOWNWARD CLOSURE). *Given a dataset $\Omega$, and a Boolean similarity function $f_S$; $f_S$ fulfills the $f_S$-downward closure if and only if (iff) $\forall O, S_1, S_2$; $O \in \Omega$; $\emptyset \neq S_1 \subseteq S_2 \subseteq R$ $[f_{S_1} freq(O) < minFreq] \Rightarrow [f_{S_2} freq(O) < minFreq]$.*

However, unlike the downward closure property for frequent itemset mining, property 1 is not always true. Its fulfillment depends on whether the frequency and the similarity function are monotonic. The monotony of the frequency and the monotonic similarity function are defined below:

PROPERTY 2 (MONOTONY OF THE FREQUENCY). *Given a dataset $\Omega$ and a Boolean similarity function $f_S$; $f_S$ fulfills the monotony of the frequency iff $\forall O, S_1, S_2; O \in \Omega$ $[\emptyset \neq S_1 \subseteq S_2 \subseteq R] \Rightarrow [f_{S_1} freq(O) \geq f_{S_2} freq(O)]$.*

DEFINITION 1 (MONOTONIC SIMILARITY FUNCTION). *Given a dataset $\Omega$ and a Boolean similarity function $f_S$; $f_S$ is non increasing monotonic iff $\forall O, O', S_1, S_2; O, O' \in \Omega$, $[\emptyset \neq S_1 \subseteq S_2 \subseteq R] \Rightarrow [f_{S_1}(O, O') \geq f_{S_2}(O, O')]$.*

The relationship between the monotony of the function, the monotony of the frequency and the $f_S$-downward closure property, shown in the following propositions 1, 2 and 3, was proofed in [29]:

Proposition 1. *If $f_S$ is a non increasing monotonic similarity function, then $f_S$ fulfills the monotony of the frequency.*

Proposition 2. *If $f_S$ fulfills the monotony of the frequency, then $f_S$ fulfills the $f_S$-downward closure.*

Proposition 3. *If $f_S$ is a non increasing monotonic similarity function, then $f_S$ satisfies the $f_S$-downward closure*

For example, the similarity function (1) is a non-increasing monotonic similarity function; then it satisfies the $f_S$-downward closure. While the similarity function (2) is not a non-increasing monotonic similarity function, then it does not satisfy the $f_S$-downward closure.

A concept used for pruning the search space of FSPs is:

Definition 2 ($f_S$-non prunable pattern). *Given a dataset $\Omega$ and a Boolean similarity function $f_S$; a subdescription $I_S(O)$, $O \in \Omega$ is an $f_S$*-non-prunable pattern *iff $I_S(O)$ is a frequent similar pattern or $I_S(O)$ is similar to another frequent similar pattern.*

In contraposition, a subdescription $I_S(O)$, $O \in \Omega$ is an *$f_S$-prunable pattern* iff $I_S(O)$ is not a frequent similar pattern and $I_S(O)$ is not similar to any frequent similar pattern.

It is important to note that, when a subdescription $I_S(O)$ is similar to another subdescription $I_S(O^{'})$ then $I_S(O)$ contributes with its repetitions in $\Omega$ to the frequency of $I_S(O^{'})$ (see equation (5)).

The following proposition, proofed in [29], allows pruning the search space by removing the subspace that contains all super descriptions of prunable patterns, without suppressing contributions to the frequency of FSPs and then without losing FSPs.

Proposition 4. *Given a dataset $\Omega$ and a non increasing monotonic Boolean similarity function $f_S$; if a subdescription $I_S(O)$ is a $f_S$*-prunable pattern*, then all superdescriptions of $I_S(O)$ are $f_S$*-prunable patterns

This paper focuses on mining all FSP using Boolean and non-increasing monotonic similarity functions. The $f_S$-downward closure and the Proposition 4 are used by our proposed algorithm, introduced in the next section.

## 4 X-FSPMINER

The algorithm proposed in this paper, named *X-FSPMiner* (Algorithm 1), consists of three general blocks: i) global variables initialization, i.e., feature value frequencies, similarities, and rank (lines 1 to 8); ii) build a novel condensed tree structure, named *FV-Tree* that store the object description collection (lines 9 to 12); iii) obtain the FSPs of only feature (1-FSPs) and mining all the frequent similar patterns from the 1-FSPs using the *FV-Tree* structure (lines 13 to 30). Details of each block are presented in the following sections.

### 4.1 Initialization block

Initialization block (Algorithm 1, lines 1 to 8) scans the object description collection in order to set and to compute the following global variables used by the subsequent blocks:

- $D'_{r_i}$: Set of values in the domain of the feature $r_i$ ($D_{r_i}$), $1 \le i \le |R|$ that occurs in the dataset $\Omega$. $D'_{r_i} \subseteq D_{r_i}$, such that, $v \in D'_{r_i}$ iff $\exists O \in \Omega$, and $O[r_i] = v$.

  Example 1. *Given the object collection $\Omega = \{O_1, \ldots, O_{10}\}$, described by a set of features $R = \{r_1, r_2, r_3\}$, such that $O_1, \ldots, O_{10}$ are defined as follows:*

**Algorithm 1:** X-FSPMiner(Ω, $f_S$, $C$, $minFreq$)

```
Input: Ω - Object collection,
       f_S - Similarity function,
       C - Comparison criteria,
       minFreq - Minimum frequency threshold
Output: F - FSP set

// ******************************************
// * Block 1: global variables initialization
// ******************************************
D′ ← initD'(Ω)
occByFV ← initOccByFV(D′)
simsByFV ← initSimsByFV(D′, C)
simOccByFV ← initSimOccByFV(occByFV, simsByFV)
nonPrunableFV ← initNonPrunableFV(simOccByFV, minFreq)
rankByFV ← initRankByFV(nonPrunableFV, occByFV)
featureFromId ← initFeatureFromId(rankByFV)
valueFromId ← initValueFromId(rankByFV)
// ******************************************
// * Block 2: build the tree structure
// ******************************************
FV-Tree ← empty FV-Tree
foreach O ∈ Ω do
    FV-Tree.addObject(O, rankByFV)
FV-Tree.updateNodeLinks()
// ******************************************
// * Block 3: mining the FSPs
// ******************************************
F ← ∅
minOcc ← minFreq ∗ |Ω|
nonPPs ← build1NonPPs(nonPrunableFV, rankByFV, occByFV, simsByFV, FV-Tree)
foreach P ∈ nonPPs do
    if P.occ + P.similarOcc ≥ minOcc then
        F.add(P)
occs ← empty array of size |FV-Tree.nodeLinks|
nPs ← empty array of size |nonPrunableFV|
while nonPPs ≠ ∅ do
    xPs ← ∅
    foreach P ∈ nonPPs do
        addExpandedPsFrom(P, xPs, occs, nPs, FV-Tree)
    foreach P ∈ xPs do
        updateSimilarsAndFrequencyOf(P, featureFromId, valueFromId, f_S, C)
        if P.occ + P.similarOcc ≥ minOcc then
            F.add(P)
    nonPPs ← getNonPPsFrom(xPs, minOcc)
return F
```

$$
\begin{aligned}
O_1 &= (1, 10, 100)\\
O_2 &= (2, 20, 200)\\
O_3 &= (2, 20, 100)\\
O_4 &= (2, 10, 100)\\
O_5 &= (3, 30, 300)\\
O_6 &= (4, 40, 400)\\
O_7 &= (6, 60, 500)\\
O_8 &= (6, 50, 500)\\
O_9 &= (5, 50, 500)\\
O_{10} &= (6, 60, 600)
\end{aligned}
$$

*Then,*

$$
\begin{array}{lrrrrrr}
D'_{r_1} = \{ & 1, & 2, & 3, & 4, & 6, & 5\}\\
D'_{r_2} = \{ & 10, & 20, & 30, & 40, & 60, & 50\}\\
D'_{r_3} = \{ & 100, & 200, & 300, & 400, & 500, & 600\}
\end{array}
$$

*We put value* $6$ *before value* $5$ *in* $D'_{r_1}$ *and value* $60$ *before value* $50$ *in* $D'_{r_2}$ *in order to note that values* $6$ *and* $60$ *appears first in Omega.*
*The position of the values in each* $D'_{r_i}$ *is used in the following global variables definitions.*

- $occByFV[i, j]$: the occurrences of each feature value pair $(r_i, v_j)$, $1 \leq i \leq |R|$, $1 \leq j \leq |D'_{r_i}|$.

Example 2. *Given the object collection* $\Omega$ *and each* $D'_{r_i}$ *defined on example 1:*

$$
occByFV = \begin{pmatrix} 1 & 3 & 1 & 1 & 3 & 1\\ 2 & 2 & 1 & 1 & 2 & 2\\ 3 & 1 & 1 & 1 & 3 & 1 \end{pmatrix}
$$

*Note that, value* $6$ *of feature* $r_1$*, is the 5th value in* $D'_{r_1}$*. Then the occurrences of the value* $6$ *is* $occByFV[1, 5] = 3$.

- $simsByFV[i, j]$: the set of features values similar to $v_j$ given a similarity function $f_S$, excluding itself, of each feature $r_i$, such that $1 \leq i \leq |R|$, $1 \leq j \leq |D'_{r_i}|$.

Example 3. *Given the object collection* $\Omega$*, and each* $D'_{r_i}$ *defined on example 1, the Boolean similarity function* $f_S$ *defined on eq. (1) and the following explicit definitions of the comparison criteria* $C_{r_1}$*,* $C_{r_2}$ *and* $C_{r_3}$*:*

| $C_{r_1}(x, y)$ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 2 | 1 | 1 | 1 | 0 | 0 | 0 |
| 3 | 0 | 1 | 1 | 1 | 0 | 0 |
| 4 | 0 | 0 | 1 | 1 | 1 | 0 |
| 5 | 0 | 0 | 0 | 1 | 1 | 1 |
| 6 | 1 | 0 | 0 | 0 | 1 | 1 |

| $C_{r_2}(x, y)$ | 10 | 20 | 30 | 40 | 50 | 60 |
|---|---|---|---|---|---|---|
| 10 | 1 | 1 | 0 | 0 | 0 | 1 |
| 20 | 1 | 1 | 1 | 0 | 0 | 0 |
| 30 | 0 | 1 | 1 | 1 | 0 | 0 |
| 40 | 0 | 0 | 1 | 1 | 1 | 0 |
| 50 | 0 | 0 | 0 | 1 | 1 | 1 |
| 60 | 1 | 0 | 0 | 0 | 1 | 1 |

| $C_{r_3}(x,y)$ | 100 | 200 | 300 | 400 | 500 | 600 |
|---|---|---|---|---|---|---|
| 100 | 1 | 1 | 0 | 0 | 0 | 1 |
| 200 | 1 | 1 | 1 | 0 | 0 | 0 |
| 300 | 0 | 1 | 1 | 1 | 0 | 0 |
| 400 | 0 | 0 | 1 | 1 | 1 | 0 |
| 500 | 0 | 0 | 0 | 1 | 1 | 1 |
| 600 | 1 | 0 | 0 | 0 | 1 | 1 |

*Then,*

$$simsByFV = \begin{pmatrix} \{2,5\} & \{1,3\} & \{2,4\} & \{3,6\} & \{1,6\} & \{4,5\} \\ \{2,5\} & \{1,3\} & \{2,4\} & \{3,6\} & \{1,6\} & \{4,5\} \\ \{2,6\} & \{1,3\} & \{2,4\} & \{3,5\} & \{4,6\} & \{1,5\} \end{pmatrix}$$

*Note that, for each feature value pair $(r_i, v_j)$, $simsByFV[i,j]$ stores the positions in $D'_{r_i}$ of the features values similar to $v_j$. For example, the feature value 10 which is represented by the pair $(r_2, v_1)$ is similar to the feature values 20 and 60, which are represented by the pairs $(r_2, v_2)$ and $(r_2, v_5)$ respectively. Then $simsByFV[2,1] = \{2,5\}$.*

- $simOccByFV[i,j]$: the occurrences, using the similarity function, of each feature value pair $(r_i, v_j)$, $1 \le i \le |R|$, $1 \le j \le |D'_{r_i}|$. $simOccByFV[i,j]$ is computed efficiently from $occByFV$ and $simsByFV$ accumulating the occurrences of $v_j$ ($occByFV[i,j]$) and the occurrences of each $v_l$ similar to $v_j$ ($occByFV[i,l]$ such that $v_l \in simsByFV[i,j]$).

Example 4. *Given the object collection $\Omega$, the similarity function $f_s$, each $D'_{r_i}$, the $occByFV$ and the $simsByFV$ defined or resulted on examples 1, 2 and 3:*

$$simOccByFV = \begin{pmatrix} 7 & 5 & 5 & 3 & 5 & 5 \\ 6 & 5 & 4 & 4 & 6 & 5 \\ 5 & 5 & 3 & 5 & 5 & 7 \end{pmatrix}$$

*Note that the occurrences, using the similarity function, of the feature value 10 (represented by the pair $(r_2, v_1)$) is:*

$$simOccByFV[2,1] = occByFV[2,1] + \sum_{l \in simsByFV[2,1]} occByFV[2,l]$$

*Variable $l$ takes values 2 and 5, which represent the feature values 20 and 60 of $r_2$ similar to the feature value 10. Then,*

$$\begin{aligned} simOccByFV[2,1] &= occByFV[2,1] + \\ &\quad occByFV[2,2] + \\ &\quad occByFV[2,5] \\ &= 2+2+2 \\ &= 6 \end{aligned}$$

- $nonPrunableFV$: Set of feature value pairs, such that, each feature value pair $(r_i, v_j)$, $1 \le i \le |R|$, $1 \le j \le |D'_{r_i}|$, is a non prunable pattern (i.e. it is a FSPs or similar to an FSPs).

Example 5. *Given the object collection $\Omega$, the similarity function $f_s$, each $D'_{r_i}$, the $simsByFV$ and the $simOccByFV$ defined or resulted on examples 1, 3 and 4. Also, given a frequency threshold $minFreq = 0.6$:*

$$nonPrunableFV = \left\{ \begin{array}{l} (1,1), (1,2), (1,5), \\ (2,1), (2,2), (2,5), (2,6), \\ (3,1), (3,5), (3,6) \end{array} \right\}$$

*Taking into account that the number of objects in $\Omega$ is 10, a pattern is considered and FSP for $minFreq = 0.6$ iff its occurrences, using the similarity function is greater than or equals to 6. Consequently only the features values 1 (represented by the pair $(r_1, v_1)$), 10 (represented by the pair $(r_2, v_1)$), 60 (represented by the pair $(r_2, v_5)$) and 600 (represented by the pair $(r_3, v_6)$) are FSPs.*
*However, the set of non prunable pattern include both, the FSPs and the patterns similar to an FSPs. Then the features values: 2 (represented by the pair $(r_1, v_2)$), 6 (represented by the pair $(r_1, v_5)$), 20 (represented by the pair $(r_2, v_2)$), 50 (represented by the pair $(r_2, v_6)$), 100 (represented by the pair $(r_3, v_1)$) and 600 (represented by the pair $(r_3, v_5)$) also are non prunable patterns.*

- $rankByFV[i, j]$: the rank position of each feature value pair $(r_i, v_j)$, such that the pair $(i, j) \in nonPrunableFV$, $1 \leq rankByFV[i, j] \leq |nonPrunableFV|$. if $(i, j) \notin nonPrunableFV$, $RankByFV[i, j] = \emptyset$. The rank position of a feature value pair $(r_i, v_j)$ is used as its id.
  We obtain the rank positions in two steps:
  (1) The non-prunable feature value pairs are ranked utilizing the occurrences from $occByFV$. The feature value pair with more occurrences is the first in the rank. If two feature value pairs have the same occurrences, the feature value pair with a less feature index is first in the rank. If there is still a tie (both feature value pairs have the same feature index), the feature value pair with a lower index is first in the rank.
  (2) The features are sorted by the sum of the rank position of its feature values. Then, the non-prunable feature value pairs are ranked again, considering the feature order first. Feature value pairs of the same feature are ranked utilizing the rank established in step 1.
  Considering only the feature values that are non-prunable patterns (i.e., are non-1-FSPs and are not similar to 1-FSPs), reduce the search space of FSPs (i.e., to reduce the number of possible combinations of feature values). Consequently, the values of the features that are prunable patterns are no longer used in successive steps of the proposed algorithm.

Example 6. *Given the object collection $\Omega$, each $D'_{r_i}$, the $occByFV$ and the $nonPrunableFV$ defined or resulted on examples 1, 2, and 5. A rank position of each feature value pair $(r_i, v_j)$ after step 1 is:*

$$rankByFV = \begin{pmatrix} 9 & 1 & \emptyset & \emptyset & 2 & \emptyset \\ 5 & 6 & \emptyset & \emptyset & 7 & 8 \\ 3 & \emptyset & \emptyset & \emptyset & 4 & 10 \end{pmatrix}$$

*Due sum by row (feature) is 12 for $r_1$, 26 for $r_2$ and 17 for $r_3$, the feature order is $r_1$, $r_3$, $r_2$. Then the rank position of each feature value pair $(r_i, v_j)$ after step 2 is:*

$$rankByFV = \begin{pmatrix} 3 & 1 & \emptyset & \emptyset & 2 & \emptyset \\ 7 & 8 & \emptyset & \emptyset & 9 & 10 \\ 4 & \emptyset & \emptyset & \emptyset & 5 & 6 \end{pmatrix}$$

- $featureFromId[id]$: the feature index $i$ of the feature value pair $(r_i, v_j)$, such that, $rankByFV[i, j] = id$.
- $valueFromId[id]$: the value index $j$ of the feature value pair $(r_i, v_j)$, such that, $rankByFV[i, j] = id$.
  $featureFromId$ and $valueFromId$ are used to decode a feature value id into a feature value pair.

Example 7. *Given the object collection $\Omega$, the $nonPrunableFV$ and the $rankByFV$ defined or resulted on examples 1, 5 and 6:*

$$featureFromId = \begin{pmatrix} 1 & 1 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 3 \end{pmatrix}$$

*and,*

$$valueFromId = \begin{pmatrix} 2 & 5 & 1 & 1 & 5 & 6 & 1 & 2 & 5 & 6 \end{pmatrix}$$

*Note that in the example the non prunable feature value* 60 *(represented by the pair* $(r_2, v_5)$*) has* 9 *as id (*$rankByFV[2, 5] = 9$*). But also a decode operation can be done from the id* 9*, using* $featureFromId[9]$ *and* $valueFromId[9]$ *to obtain* 2 *as feature index and* 5 *as value index.*

## 4.2 Build the tree structure

To condense a collection Ω of objects described by numerical and non-numerical features and to compute the frequency of the FSPs, we propose the *FV-Tree* structure, which is used by our novel algorithm X-FSPMiner (Algorithm 1).

*FV-Tree* is a tree structure in which each branch from the root to a leaf represents an object $O \in \Omega$. Each node ($FV\text{-}Node$) in a branch (except the root node) represents a feature value $O[r_i] = (r_i, v_j)$, $1 \le i \le |R|$, $1 \le j \le |D'_{r_i}|$, $(i, j) \in nonPrunableFV$. Only non prunable feature values are considered and objects with one o more prunable feature values are represented in a branch from the root but not necessarily to a leaf.

Each $FV\text{-}Node$ contains:

- $fvId$: Identifier of the feature value. $fvId = rankByFV[i, j]$.
- $occ$: Number of occurrences in Ω of the subdescription that contains all ancestor feature values and itself (i.e., all feature values in the branch from the root to itself).
- $preCode$ : Pre-order of the FV-Node in the *FV-Tree*.
- $parent$: link to the parent FV-Node in the *FV-Tree*.
- $children$: Set of links to the children $FV\text{-}Nodes$ in the *FV-Tree*.

To quickly know if an $FV\text{-}Node$ is a child, and to quickly access it, we use a Self-balancing binary search tree (SBBST) as the set of children. The $fvId$ of each $FV\text{-}Node$ child is used as the key of the nodes of the SBBST.

The root of an *FV-Tree* is an special $FV\text{-}Node$, without $fvId$, $occs$, $preCode$, $postCode$ and $parent$. It only contains $children$. *FV-Tree* also includes an array $nodesLinks$ with links to the all the $FV\text{-}Nodes$.

Starting from an empty *FV-Tree*, the special root FV-Node is created. Later, each object $O \in \Omega$ is added to the *FV-Tree*. The method to add an object to the *FV-Tree* (Procedure $FV\text{-}Tree$.addObject) first puts the ids of the feature values of the object that there are in the rank (i.e., $f_S$-frequent feature values or non $f_S$-frequent feature values similar to an $f_S$-frequent feature value) into the list $fvIds$ (lines 1 to 5). Also, the list $fvIds$ is sorted according to the rank (line 6). Besides, an auxiliary $FV\text{-}Node$, $auxFVNode$, is positioned at the root of the *FV-Tree* (line 7). $auxFVNode$ is used to move (following the rank order) over the branch containing the object's feature values to add (lines 8 to 19). If there is an $FV\text{-}Node$ in the branch with the same $fvId$ that the *id* of the corresponding feature value of the object, the occurrences of this $FV\text{-}Node$ is increased by 1 (lines 9 to 11). Otherwise, a new $FV\text{-}Node$, for the corresponding feature value of the object, is created, initialized, and inserted in the branch (lines 12 to 18).

After all objects in Ω are added to the *FV-Tree*, $preCode$ of each $FV\text{-}Node$ in *FV-Tree* (except for the root $FV\text{-}Node$) is set by traversal the *FV-Tree* in pre-order. Later, the $nodesLinks$ array is created and each $nodesLinks[i]$ is updated with a link to the $FV\text{-}Node$, such that, $FV\text{-}Node.preCode = i$.

Example 8. *Given the object collection Ω, the* $nonPrunableFV$ *and the* $rankByFV$ *defined or resulted on examples 1, 5 and 6.*

*Figure 2 show the* FV-Tree *structure after insert each object.* $FV$*-nodes are represented by rectangles that contains* $fvId$ *and* $occ$*. Links to parent and children are represented by black arrows and dashed arrows, respectively. Dark gray* $FV$*-nodes represent the* $FV$*-nodes created or updated at the insertion of the object. The bold* $occ$ *value indicates*

**Procedure** *FV-Tree*.addObject($O$, $rankByFV$)

```
Input: O - Object ∈ Ω,
       rankByFV - Rank by feature value

fvIds ← empty List
for i ← 1 to |R| do
    v_j = O[r_i]
    if ∃ rankByFV[i, j] then
        fvIds.add(rankByFV[i, j])
sort(fvIds)
auxFVNode ← root FVNode
foreach id ∈ fvIds do
    if auxFVNode.children.contains(id) then
        child ← auxFVNode.children.get(id)
        child.occ ← child.occ + 1
    else
        child ← empty FVNode
        child.fvId ← id
        child.parent ← auxFVNode
        child.occ ← 1
        child.children ← empty SBBST
        auxFVNode.children.add(child)
    auxFVNode ← child
```

*that the associated FV-nodes already existed before the insertion of an object, and then only the occurrence of the node was updated.*

*After insert into the* FV-Tree *the* 10 *objects in* $\Omega$*, precode is set for each FV-node and the nodeLinks array is created. Figure 3 show the final* FV-Tree *structure.*

## 4.3 Mining the frequent similar patterns

The general idea for mine all FSPs (Algorithm 1, lines 13 to 30) follows a breadth-first search strategy, which starts from 1-non prunable patterns, i.e., non-prunable patterns with only one feature value (line 15), and the 1-FSPs are extracted (lines 16 to 18). On subsequent steps (lines 19 to 30): expanding the non-prunable patterns by adding 1-non prunable patterns (lines 22 to 24), computing the frequency of the generated patterns (lines 25 to 26), and The FSPs (lines 27 to 28) and the non-prunable patterns (line 29) are extracted from the set of generated patterns.

The *FV-Tree* structure built on the previous block and a novel structure named *SP-Node* to represent each pattern are used in the expansion process and the computation of the frequency of the generated patterns. *SP-Node* is a structure representing a pattern and includes other relevant information to achieve a fast expansion process and the computation of its frequency. An *SP-Node* contains:

- $fvIds$: array of the ids of the feature values that compound the pattern. Using the global variables $featureFromId$ and $valueFromId$ can be decoded each id into a feature value pair $(r_i, v_j)$. $fvIds$ follows the reverse rank order.

Example 9. *Given the object collection* $\Omega$*, the RankByFV and the* FV-Tree *defined or resulted on examples 1, 6 and 8. The pattern* $(10, 100)$ *in* $\Omega$ *is represent by an SP-Node, such that:*

$$SP\text{-}Node.fvIds = (7\ 4)$$

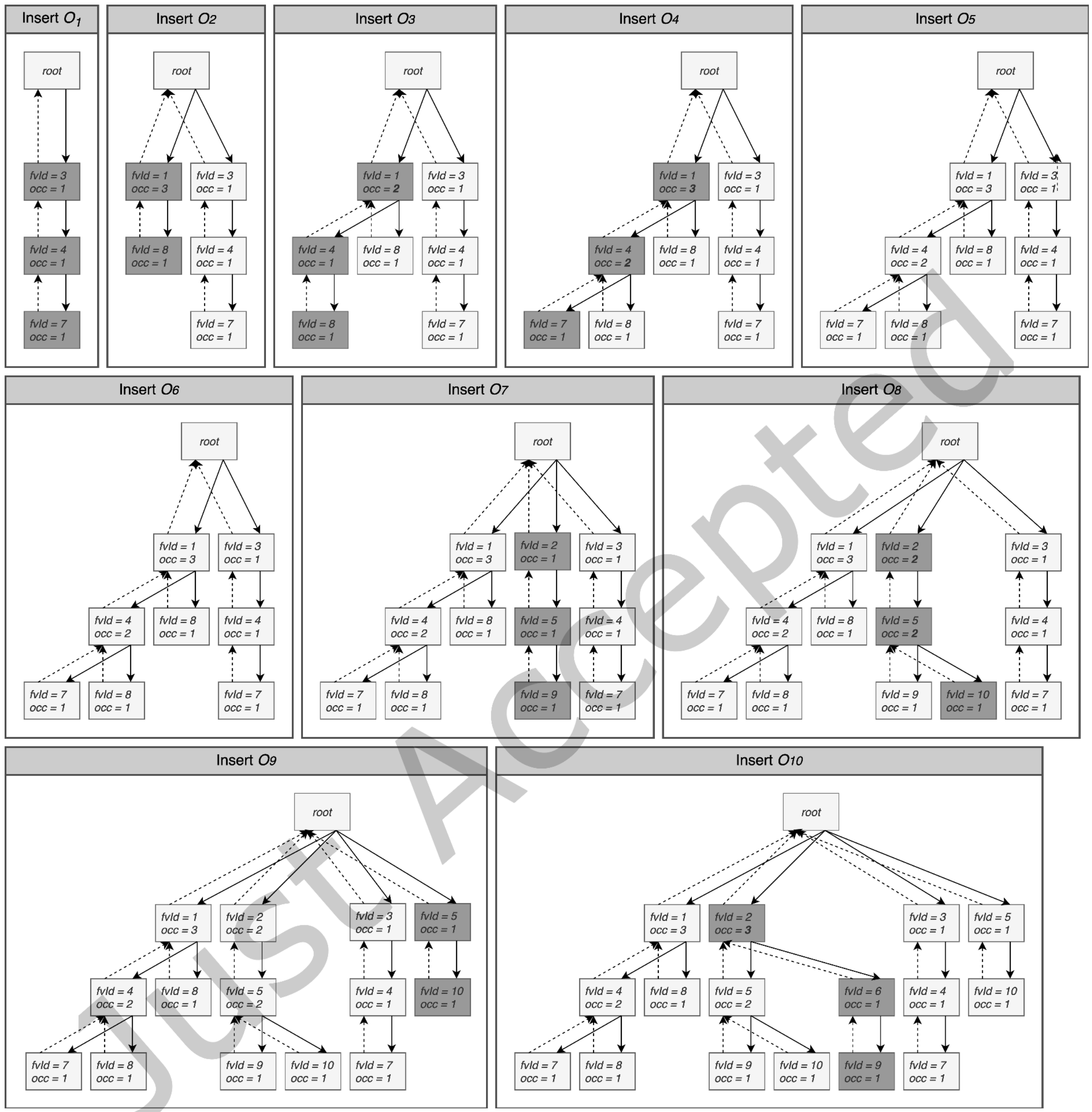


Fig. 2. *FV-Tree* after insert each object of the collection $\Omega$ defined in example 1



*Remember that the feature value* 100 *is represented by the feature value pair* $(3, 1)$ *whose id is* $RankByFV[3, 1] = 4$*; and the feature value* 10 *is represented by the feature value pair* $(1, 1)$ *whose id is* $RankByFV[1, 1] = 7$.

- $patternSubTrees$: array of pairs $(preCode, occ)$. Each $preCode$ represents the $FV\text{-}Node$ in *FV-Tree*, such that, $FV\text{-}Node.fvId = fvIds[|fvIds|]$, and the subtree from $FV\text{-}Node$ contains at least 1 occurrence of the

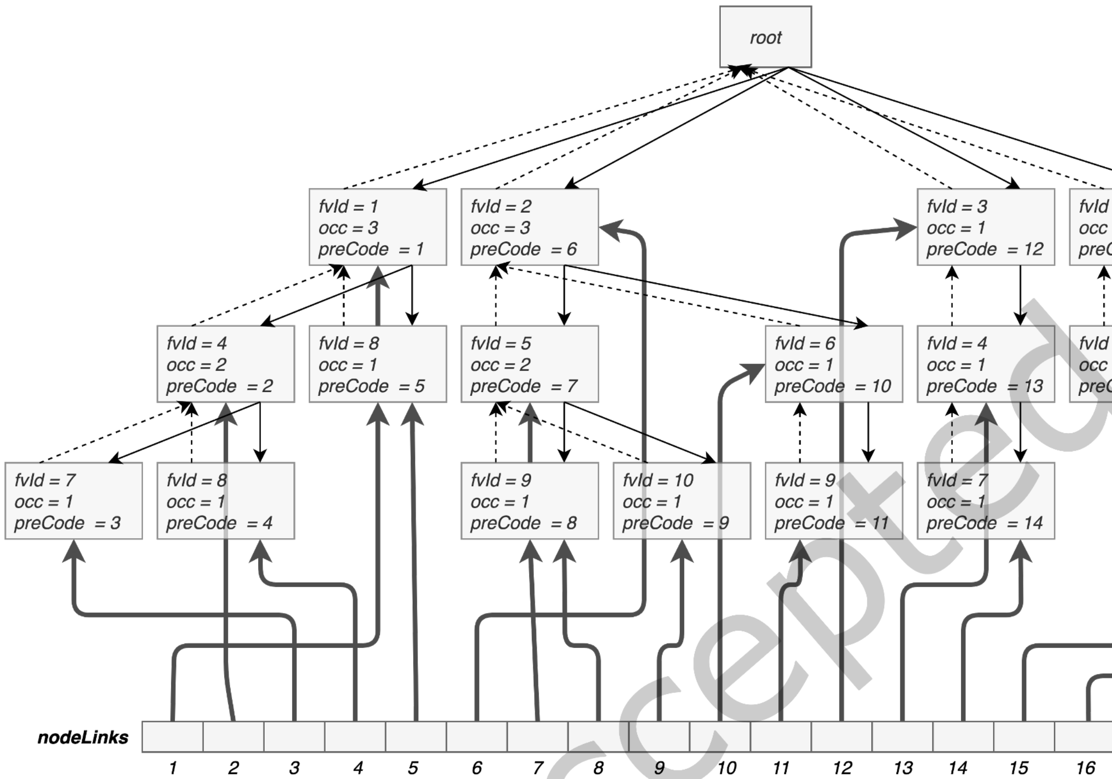


Fig. 3. Final *FV-Tree* that contains the objects of the collection Ω defined in example 1



pattern. Each $occ$ of a pair $(preCode, occ)$ is the number of occurrences of the patterns in the subtree from $FV\text{-}Node$.

Example 10. *From previous example 9:*

$$SP\text{-}Node.patternSubTrees = (\ (2,\ 1)\ (13,\ 1)\ )$$

*Note that there are only two $FV$-Nodes in* FV-Tree*, such that, $FV\text{-}Node.fvId = 4$ (equal to the last component of the $fvIds$). These $FV$-Nodes have $preCodes$ 2 and 13. Also, each one is the root of a subtree that contains at least 1 occurrence of the pattern.*

- $occ$: number of repetitions of the pattern. It is the sum all occurrences from the pairs $(preCode, occ)$ in $patternSubTrees$.
- $similarPatterns$: an array of links to the similar patterns represented as *SP-Nodes*
- $similarOcc$: number of repetitions of similar patterns, but not the same pattern.
  Note that $\frac{occ+similarOcc}{|\Omega|}$ is the *frequency* of the pattern.
- *producer*: link to another pattern represented as an *SP-Node* from which the pattern was generated by adding a 1-non prunable pattern.
- *expansions*: Set of links to patterns represented by *SP-Nodes* produced by adding a 1-non-prunable pattern. To quickly know if an *SP-Node* is in *expansions*, but also quickly know if there is at least one *SP-Node* in *expansions* with a particular last feature, and to access it quickly, we use a bi-level tree of Self-balancing

binary search tree (SBBST) as the set of expansions. The root is an SBBST that uses as the keys, the feature id of the last feature of each $SP - Node$ in $expansions$ ($featureFromId[SP - Node.fvIds[|fvIds|]]$). The content of the nodes of the SBBST are SBBSTs that use as the keys the value id of the last feature of each $SP - Node$ in $expansions$ ($valueFromId[SP - Node.fvIds[|fvIds|]]$). This structure is used to obtain the $similarPatterns$ of the $SP$-$Nodes$ produced by adding a 1-non prunable pattern.

The proposed algorithm (Algorithm 1) delegates the build of the 1-non prunable patterns set to the Function build1NonPPs. It starts creating an empty array $nonPP$ of size $|nonPrunableFV|$ to put the $SP$-$Node$ that represents the 1 non prunable patterns (line 1). In a first Loop (lines 2 to 9), for each feature value pair $(r, v) \in nonPrunableFV$ an $SP$-$Node$ is build and putted in $nonPP$, initializing the most of the attributes ($fvIds$, $occ$, $producer$, $expansions$ and $patternSubTrees$) using the global variables. In a second loop (lines 10 to 16), the $SP$-$Nodes$ built are revisited and the attributes $similarOcc$ and $similarPatterns$ are initialized. Finally (lines 17 to 18), the $FV$-$Tree$ is traversed to add to the $patternSubTrees$ attribute of the corresponding $SP$-$Nodes$ the pair ($precode, occ$) of each $FV$-$Node$ in $FV$-$Tree$ by means of the recursive Procedure updatePsSubTrees.

**Function** build1NonPPs($nonPrunableFV$, $rankByFV$, $occByFV$, $simsByFV$, $FV$-$Tree$)

**Input:** $nonPrunableFV$ - Non prunable feature values,
$rankByFV$ - Rank by feature value,
$occByFV$ - Occurrence by feature value,
$simsByFV$ - Similars by feature value,
$FV$-$Tree$ - Tree structure
**Output:** $nonPPs$ - 1-non prunable patterns set

1 $nonPPs \leftarrow$ empty array of size $|nonPrunableFV|$
2 **for** $(r, v) \in nonPrunableFV$ **do**
3     $P \leftarrow$ empty $SP$-$Node$
4     $P.fvIds \leftarrow (\ rankByFV[r, v]\ )$
5     $P.occ \leftarrow occByFV[r, v]$
6     $P.producer \leftarrow \emptyset$
7     $P.expansions \leftarrow$ empty $SBBST$
8     $P.patternSubTrees = \leftarrow \emptyset$
9     $nonPPs[rankByFV[r, v]] \leftarrow SP$
10 **for** $(r, v) \in nonPrunableFV$ **do**
11     $P \leftarrow nonPPs[rankByFV[r, v]]$
12     $P.similarOcc \leftarrow 0$
13     **for** $sv \in simsByFV[r, v]$ **do**
14         $sP \leftarrow nonPPs[rankByFV[r, sv]]$
15         $P.similarPatterns.add(sP)$
16         $P.similarOcc \leftarrow P.similarOcc + sP.occ$
17 **foreach** $child \in FV$-$Tree.root$ **do**
18     updatePsSubTrees($nonPPs$, $child$)
19 **return** $nonPPs$

After building the 1-non-prunable patterns, the Algorithm 1 filters the FSPs from $nonPP$ into $F$ (lines 16 to 18). Also, two large arrays ($occs$ and $nPs$) systematically used during the expansion process are initialized to avoid multiple inner initializations (lines 19 and 20). The main loop of the proposed algorithm (Algorithm 1 lines 21 to 29), is executed while there are new non prunable patterns ($nonPPs \neq \emptyset$). On each iteration, the expansions by adding one more feature value of the currents non-prunable patterns are generated (lines 22 to 24). The expansion

**Procedure** updatePsSubTrees($nonPPs$, $root$)

```
Input: root - FV-Node ∈ FV-Tree,
       nonPPs - array of SP − Nodes

foreach child ∈ root.children do
    P ← nonPPs[child.fvId]
    P.patternSubTrees.add( (child.preCode, child.occ) )
    updatePsSubTrees(nonPPs, child)
```

of each non-prunable pattern is delegated to the Procedure addExpandedPsFrom. When all expanded patterns ($xPs$) are generated, for each one (lines 25 to 28), the set of links to its similar patterns($similarPatterns$), and its number of repetitions of similar patterns $similarOcc$ is updated. Also, each expanded pattern, that is an FSP, is added to $F$. The update of the $similarPatterns$ and $similarOcc$ attributes of each expanded pattern is delegated to the Procedure updateSimilarsAndFrequencyOf. The last step in an iteration is to filter the non-prunable patterns from the expanded patterns (line 29), which is delegated to the Function getNonPPsFrom, and to substitute the old set of non-prunable patterns (*nonPP*) by the newest set. Finally, after the main loop, the mined FSP are returned (line 30).

The way to generate the expansions of a pattern (Procedure addExpandedPsFrom) and how to calculate its frequency (Procedure updateSimilarsAndFrequencyOf) are the two critical processes of the proposed algorithm *X-FSPMiner* that make the difference in terms of execution time respect to the related work algorithms that mine all the FSPs.

The process to generate the expansions of a pattern (Procedure addExpandedPsFrom) follows two main ideas. The first one is that each possible expansion generated is a valid expansion in $\Omega$. That is, to generate directly, without candidate expansions generation and filtering the ones in $\Omega$. The second one is to add only features values that are non-prunable patterns to the pattern. This idea reduces the number of candidates to frequent similar patterns because if the new feature value is a prunable pattern, the expanded pattern will also be. The use of the *FV-Tree* built on block 2 allows developing both ideas. *FV-Tree* only contains in its branches patterns that appear in $\Omega$, but also it does not contain prunable features values.

Given a pattern $P$, Procedure addExpandedPsFrom starts from an empty set (*nodeIds*) of ids of $FV$-$Nodes$ (line 1). In a first Loop (lines 2 to 11), the nodes in *FV-Tree* which represent the root of a subtree that contains at least one occurrence of the pattern $P$ are obtained. The *preCode* to obtain each *node* is extracted from the pairs $(preCode, occ) \in P.patternSubTrees$ (line 3). The ancestors of each *node* are traversed (lines $4 - 10$). Each ancestor node represents a feature value that, together with the features values of the $P$, will make up a new valid expanded pattern. Since two o more *nodes* can have the same ancestor node, an ancestor node represents the root of a subtree that contains exactly the sum of the occurrences of the pattern $P$ in the corresponding subtrees whose root is *node*, as the occurrences of the pattern expanded pattern. In order to directly identify if an ancestor node was previously visited and to update the occurrences of the expanded pattern that it represents, the *occs* array is used (lines 7 to 10). Also the *preCode* of the ancestor nodes visited are added to the set *nodeIds* (line 11).

From an empty set ($fvIds$) of ids of features values (line 12), in a second loop (lines 13 to 31), the feature value id ($fvId$) of each ancestor node (with *nId* as *preCode*) is obtained. Two o more ancestor nodes can have the same $fvId$ (i.e., two o more ancestor nodes can represent the same expanded pattern). In order to directly identify if an $fvId$ was previously visited and the corresponding expanded pattern was built, the *nPs* array is used (line 15). If the $fvId$ was previously visited, the expanded pattern *nP* is recovered and the id of each ancestor node (*nId*) and the occurrences of the expanded pattern in the corresponding ancestor node subtree ($occs[nid]$) are added to the attribute $patternSubTrees$ of the expanded pattern (lines 15 to 18). Otherwise (lines 19 to 31), the

**Procedure** addExpandedPsFrom($P$, $xPs$, $occs$, $nPs$, $FV\text{-}Tree$)

```
Input: P - a non prunable pattern,
       xPs - expanded patterns set,
       occs - empty array of size |FV-Tree.nodeLinks|,
       nPs - empty array of size |nonPrunableFV|,
       FV-Tree - Tree structure

nodeIds ← ∅
foreach (preCode, occ) ∈ P.patternSubTrees do
    node ← FV-Tree.nodeLinks[preCode]
    while node.parent ≠ FV-Tree.root do
        node ← node.parent
        nId ← node.preCode
        if ∃ occs[nId] then
            occs[nId] ← occs[nId] + occ
        else
            occs[nId] ← occ
            nodeIds.add(nId)
fvIds ← ∅
foreach nId ∈ nodeIds do
    fvId ← nodeLinks[nId].fvId
    if ∃ nPs[fvId] then
        nP ← nPs[fvId]
        nP.occ ← nP.occ + occs[nId]
        nP.patternSubTrees.add( (nId, occs[nId]) )
    else
        nP ← empty SP-Node
        nP.fvIds ← empty array of size |P.fvIds| + 1
        for i = 1 to |P.fvIds| do
            nP.fvIds[i] ← P.fvIds[i]
        nP.fvIds[|nP.fvIds|] ← fvId
        nP.occ ← occs[nId]
        nP.producer ← P
        P.expansions.add(nP)
        nP.expansions ← empty SBBST
        nP.patternSubTrees ← {(nId, occs[nId])}
        nPs[fvId] ← nP
        xPs.add(nP)
foreach nId ∈ nodeIds do
    occs[nId] ← ∅
foreach fvId ∈ fvIds do
    nPs[fvId] ← ∅
```

expanded pattern is built and the most of its attributes ($fvIds$, $occ$, $producer$, $expansions$ and $patternSubTrees$) are initialized. Also, the expanded pattern $nP$ is assigned to $nPs[fvId]$ and added to the set of expanded patterns.

Since few positions of the arrays $occs$ and $nPs$ are used in each call to the algorithm, and they are large arrays, these arrays are not initialized each time but a single time by the caller (i.e., by the X-FSPMiner algorithm).

Consequently, the positions used by the procedure in the arrays *occs* and *nPs* are cleared for using it by subsequent procedure calls (lines 32 to 35).

**Procedure** updateSimilarsAndFrequencyOf($P$, $featureFromId$, $valueFromId$, $f_S$, $C$)

```
Input: P - expanded pattern,
       featureFromId - Array of the feature id for each feature value id,
       valueFromId - Array of the value id for each feature value id,
       f_S - Similarity function,
       C - Comparison criteria

1  similarCandidates ← ∅
2  fvId ← P.fvIds[|P.fvIds|]
3  fId ← featureFromId[fvId]
4  vId ← valueFromId[fvId]
5  foreach prodSP ∈ P.producer.similarPatterns do
6      if prodSP.expansions ≠ ∅ then
7          xPs ← prodSP.expansions.getSubset(fId)
8          foreach xP ∈ xPs do
9              fvIdxP ← xP.fvIds[|xP.fvIds|]
10             vIdxP ← valueFromId[fvIdxP]
11             if C_{r_fId}(v_vId, v_vIdxP) = 1 then
12                 similarCandidates.add(xP)
13 xPs ← P.producer.expansions.getSubset(fId)
14 foreach xP ∈ xPs do
15     fvIdxP ← xP.fvIds[|xP.fvIds|]
16     if fvIdxP ≠ fvId then
17         vIdxP ← valueFromId[fvIdxP]
18         if C_{r_fId}(v_vId, v_vIdxP) = 1 then
19             similarCandidates.add(xP)
20 P.similarOcc ← 0
21 foreach sP ∈ similarCandidates do
22     if f_S(P, sP) = 1 then
23         P.similarOcc ← P.similarOcc + sP.occ
24         P.similarPatterns.add(sP)
```

Given a pattern $P$, the process to update the set of its similar patterns and to calculate its frequency (Procedure updateSimilarsAndFrequencyOf) takes advantage of the following two consequences of the non-increasing monotonic similarity function definition. The first one is that only expansions of patterns similar to the pattern expanded to obtain $P$ can be similar to $P$. The second one is that only patterns that have similar values of the last features can be similar patterns.

Procedure updateSimilarsAndFrequencyOf starts from an empty set (*similarCandidates*) of candidate patterns to be similar to $P$ (line 1). Later (lines 2 to 4), the feature value id ($fvId$), the feature id ($fId$) and the value id ($vId$) of the last feature value of the pattern $P$ is obtained. In a first loop (lines 5 to 12), the expansions of patterns that are similar to the pattern that was expanded to build $P$ and have the last feature similar to the last feature of $P$ are added to the set *similarCandidates*. Since the attribute, *expansions* of a pattern is a bi-level tree of the Self-balancing binary search tree, the expanded patterns with the same last feature are efficiently obtained from each pattern, similar to the pattern that was expanded to build $P$ (line 7). In a second loop (lines 14 to 19), the expansion patterns of the pattern that was expanded to build $P$ have the same last feature that $P$ and their values of the last feature are similar, are added to the set $similarCandidates$.

In a second loop (lines 14 to 19), the expansion patterns of the pattern that was expanded to build $P$ *and* have the same last feature that $P$, *with* their values are similar, are added to the set $similarCandidates$.

Finally (lines 20 to 24), in a third loop, the similarity between $P$ and each candidate patterns to be similar to $P$ is evaluated. Then, patterns similar to $P$ are added to $P.similarPatterns$ and the attribute $P.similarOcc$ is updated.

**Function** getNonPPsFrom($xPs$, $minOcc$)

**Input:** $xPs$ - expanded patterns set,
$minOcc$ - Minimum occurrence threshold
**Output:** $nonPPs$ - non prunable patterns set

```
1  nonPPs ← ∅
2  foreach P ∈ xPs do
3      if P.occ + P.similarOcc ≥ minOcc then
4          nonPPs.add(P)
5      else
6          foreach sP ∈ P.similarPatterns do
7              if sP.occ + sP.similarOcc ≥ minOcc then
8                  nonPPs.add(P)
9                  break
10 return nonPPs
```

The filtering of the non prunable patterns from the expanded patterns (Function getNonPPsFrom) starts from an empty new non prunable pattern set *nonPP* (line 1) and simply adds to the expanded patterns that are FSPs (lines 2 to 4) or have a least one similar patterns that is an FSP (lines 5 to 9). The resulted *nonPP* is returned (line 10).

The exclusion of the prunable patterns saves time consumed to generate expanded patterns that will not be FSP, nor will they contribute to the frequency of the FSP and compute their frequencies. A prune is done without losing possible FSPs.

## 5 EXPERIMENTS AND RESULTS

In this section, the performance of the proposed algorithm *X-FSPMiner* is evaluated and compared with the state-of-the-art algorithms that mine all FSPs using Boolean and non-increasing monotonic similarity functions (i.e., *ObjectMiner* and *STreeDC-Miner*). Since these algorithms mine the same set of FSPs, the comparison was made in terms of the efficiency of the algorithm measuring their runtimes. The less runtime of an algorithm is, the more efficient the algorithm is.

Table 1 describes the datasets used in the experiment. These datasets proceed from the UC Irvine Machine Learning Repository[1].

The *X-FSPMiner* algorithm was implemented in Java. The Java implementations of *ObjectMiner* and *STreeDC-Miner* provided by their authors were used. The experiment was done on a workstation with an Intel(R) Xeon(R) CPU E5-2603 v3 at 1.60 GHz and 96Gb of RAM.

[1]https://archive.ics.uci.edu/ml/index.php

Table 1. Description of datasets.

| **Datasets** | **Objects** | **Non-numerical Features** | **Numerical Features** |
|---|---|---|---|
| *Abalone* | 4177 | 2 | 7 |
| *Auto MPG* | 392 | 3 | 5 |
| *AutoUniv au6* | 1000 | 4 | 36 |
| *Balance Scale* | 576 | 1 | 4 |
| *Breast Cancer Wisconsin* | 683 | 1 | 9 |
| *Liver disorders* | 345 | 1 | 6 |
| *Car Evaluation* | 1728 | 5 | 2 |
| *Contraceptive Method Choice* | 1473 | 9 | 1 |
| *Credit Approval* | 690 | 9 | 7 |
| *Pima indians diabetes* | 768 | 1 | 8 |
| *Glass Identification* | 146 | 1 | 9 |
| *Heart Disease* | 270 | 1 | 13 |
| *Indian Liver Patient* | 579 | 2 | 9 |
| *Iris* | 150 | 1 | 4 |
| *Metadata* | 528 | 2 | 17 |
| *Poker Hand* | 1000000 | 11 | 0 |
| *Teaching Assistant Evaluation* | 151 | 3 | 3 |
| *Vehicle Silhouettes* | 846 | 1 | 18 |
| *Wine* | 178 | 1 | 13 |

Table 2. Number of FSPs by varying the *minFreq* from 0.02 to 0.20 with step 0.02 for each dataset.

| **Datasets** | **0.02** | **0.04** | **0.06** | **0.08** | **0.10** | **0.12** | **0.14** | **0.16** | **0.18** | **0.20** |
|---|---|---|---|---|---|---|---|---|---|---|
| *Abalone* | 1233035 | 795360 | 492615 | 305439 | 187624 | 106475 | 62088 | 32168 | 15542 | 8724 |
| *Auto MPG* | 17422 | 7884 | 4290 | 2489 | 1622 | 1105 | 853 | 569 | 396 | 250 |
| *AutoUniv au6* | 657525 | 148507 | 54242 | 24580 | 13162 | 7313 | 4359 | 2837 | 1839 | 1369 |
| *Balance Scale* | 342 | 142 | 54 | 46 | 42 | 38 | 30 | 26 | 22 | 10 |
| *Breast Cancer Wisconsin* | 6080 | 3329 | 1991 | 1535 | 1311 | 1039 | 722 | 466 | 329 | 277 |
| *Liver disorders* | 14493 | 8466 | 5529 | 3908 | 2764 | 2040 | 1529 | 1170 | 896 | 703 |
| *Car Evaluation* | 1606 | 403 | 303 | 220 | 86 | 44 | 44 | 40 | 36 | 31 |
| *Contraceptive Method Choice* | 8573 | 2588 | 1211 | 727 | 475 | 326 | 233 | 185 | 145 | 121 |
| *Credit Approval* | 11000118 | 6623427 | 4624674 | 3429405 | 2678886 | 2081617 | 1642501 | 1306608 | 1043208 | 852004 |
| *Pima indians diabetes* | 45392 | 21941 | 12474 | 7974 | 5181 | 3483 | 2580 | 1938 | 1415 | 1094 |
| *Glass Identification* | 75282 | 50310 | 36861 | 27445 | 20326 | 15456 | 11842 | 8935 | 6811 | 5297 |
| *Heart Disease* | 216199 | 71701 | 28352 | 14805 | 8434 | 4633 | 2966 | 1808 | 1231 | 882 |
| *Indian Liver Patient* | 391910 | 244431 | 174912 | 131207 | 103625 | 82719 | 67913 | 56948 | 47468 | 41623 |
| *Iris* | 1905 | 1011 | 549 | 320 | 207 | 109 | 82 | 54 | 44 | 41 |
| *Metadata* | 62535248 | 60037634 | 1107573 | 1107565 | 307308 | 307308 | 149632 | 149632 | 138858 | 89130 |
| *Poker Hand* | 739 | 288 | 247 | 62 | 62 | 42 | 22 | 22 | 22 | 22 |
| *Teaching Assistant Evaluation* | 1840 | 847 | 497 | 330 | 221 | 167 | 113 | 78 | 53 | 35 |
| *Vehicle Silhouettes* | 3360285 | 499974 | 158384 | 69829 | 34244 | 18681 | 10730 | 6777 | 5079 | 3756 |
| *Wine* | 33042 | 8761 | 4369 | 2298 | 1559 | 1021 | 783 | 566 | 411 | 285 |

For all dataset as a non-Boolean and non-increasing monotonic similarity function, we use the similarity function (1) of section 3. For each numerical feature $r$, we use the comparison criteria (4) with $\varepsilon = \alpha \times (maxV - minV)$, where $maxV$ is the maximum $v \in D_r$, $minV$ is the minimum $v \in D_r$ and $\alpha = 0.05$. In the particular case of numerical features where $maxV - minV \leq 5$ we use $\alpha = 0.20$. For non-numerical features, we use the equality (3) as comparison criteria.

The experiment carried out consisted on the execution of the proposed algorithm *X-FSPMiner* and the state-of-the-art algorithms *ObjectMiner* and *STreeDC-Miner* by varying the *minFreq* from 0.02 to 0.20 with step size of 0.02 for each dataset. In each execution of an algorithm, the number of FSPs mined and the runtime was measured [2].

The compared algorithms, including the proposed *X-FSPMiner* algorithm, mine the same FSPs (the set of all FSPs given a minFreq threshold). The number of mined FSPs is shown in Table 2. The less the *minFreq* threshold is, the greater the set of all FSPs is. But also, the runtime consumed by each algorithm (albeit different) is greater.

[2]The datasets, the algorithms, a script to exec the experiment, and the raw results are available on https://drive.google.com/drive/folders/1XxZtayzcsZdVZRO8QzoJo30ntYBs-ZV_

Table 3. Runtimes (in millisecs) of the algorithms *ObjectMiner*, *STreeDC-Miner*, and *X-FSPMiner* by varying the $minFreq$ from 0.02 to 0.20 with step 0.02 for each dataset.

| **Datasets** | **Algorithms** | **0.02** | **0.04** | **0.06** | **0.08** | **0.10** | **0.12** | **0.14** | **0.16** | **0.18** | **0.20** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *Abalone* | *ObjectMiner* | 1321836 | 1112347 | 921384 | 782100 | 680862 | 569590 | 486949 | 419882 | 349935 | 195390 |
| | *STreeDC-Miner* | 225719 | 225434 | 200320 | 167143 | 128258 | 91194 | 63358 | 40803 | 21199 | 11148 |
| | *X-FSPMiner* | **110589** | **111646** | **106172** | **92077** | **77091** | **58944** | **41561** | **28913** | **15447** | **8351** |
| *Auto MPG* | *ObjectMiner* | 1376 | 1188 | 889 | 903 | 665 | 627 | 460 | 350 | 359 | 264 |
| | *STreeDC-Miner* | 539 | 449 | 394 | 337 | 304 | 281 | 301 | 230 | 234 | 192 |
| | *X-FSPMiner* | **421** | **332** | **285** | **241** | **223** | **205** | **204** | **188** | **178** | **137** |
| *AutoUniv au6* | *ObjectMiner* | 527636 | 314719 | 230808 | 148876 | 107343 | 68996 | 46281 | 31440 | 18273 | 11923 |
| | *STreeDC-Miner* | 78216 | 39737 | 21985 | 15110 | 11125 | 8952 | 7152 | 6524 | 6299 | 5349 |
| | *X-FSPMiner* | **58908** | **25660** | **13973** | **8178** | **5803** | **4090** | **3002** | **2298** | **1951** | **1561** |
| *Balance Scale* | *ObjectMiner* | **65** | **56** | **52** | **51** | **51** | **51** | **52** | **52** | **55** | **44** |
| | *STreeDC-Miner* | 131 | 111 | 121 | 106 | 123 | 114 | 106 | 121 | 100 | 92 |
| | *X-FSPMiner* | 78 | 89 | 85 | 82 | 66 | 82 | 81 | 82 | 80 | 56 |
| *Breast Cancer Wisconsin* | *ObjectMiner* | 244 | 236 | 187 | 142 | 136 | 127 | 131 | 118 | 106 | 105 |
| | *STreeDC-Miner* | 391 | 187 | 218 | 161 | 157 | 149 | 182 | 136 | 170 | 153 |
| | *X-FSPMiner* | **184** | **126** | **135** | **111** | **100** | **97** | **92** | **86** | **84** | **98** |
| *Liver disorders* | *ObjectMiner* | 720 | 520 | 487 | 368 | 363 | 299 | 270 | 253 | 238 | 233 |
| | *STreeDC-Miner* | 480 | 433 | 391 | 313 | 371 | 341 | 333 | 330 | 280 | 299 |
| | *X-FSPMiner* | **340** | **285** | **275** | **226** | **241** | **230** | **224** | **200** | **195** | **183** |
| *Car Evaluation* | *ObjectMiner* | 160 | **121** | 118 | 115 | 99 | **111** | **95** | **95** | 96 | 96 |
| | *STreeDC-Miner* | 201 | 164 | 174 | 126 | 148 | 153 | 120 | 148 | 150 | 148 |
| | *X-FSPMiner* | **128** | 136 | **110** | **102** | **97** | 118 | 98 | 104 | **95** | **95** |
| *Contraceptive Method Choice* | *ObjectMiner* | 461 | 267 | 262 | 183 | 198 | 145 | 131 | 148 | 122 | 117 |
| | *STreeDC-Miner* | 353 | 242 | 175 | 202 | 149 | 144 | 137 | 145 | 165 | 163 |
| | *X-FSPMiner* | **322** | **156** | **139** | **147** | **117** | **111** | **103** | **113** | **103** | **100** |
| *Credit Approval* | *ObjectMiner* | 1281021 | 624057 | 415166 | 311993 | 250827 | 209661 | 177787 | 164093 | 136431 | 125857 |
| | *STreeDC-Miner* | 837825 | 544999 | 450406 | 370530 | 325499 | 287746 | 260313 | 236854 | 214828 | 194065 |
| | *X-FSPMiner* | **678452** | **373450** | **275090** | **221493** | **178267** | **154294** | **123777** | **116761** | **102169** | **83839** |
| *Pima indians diabetes* | *ObjectMiner* | 9371 | 7211 | 6223 | 5362 | 4587 | 4104 | 3906 | 3254 | 3063 | 2564 |
| | *STreeDC-Miner* | 2025 | 1608 | 1361 | 1346 | 1113 | 993 | 986 | 893 | 815 | 778 |
| | *X-FSPMiner* | **1293** | **1168** | **918** | **903** | **673** | **794** | **619** | **586** | **480** | **459** |
| *Glass Identification* | *ObjectMiner* | 2164 | 1463 | 1381 | 1019 | 1069 | 925 | 812 | 737 | 635 | 675 |
| | *STreeDC-Miner* | 941 | 979 | 970 | 715 | 575 | 499 | 496 | 472 | 420 | 425 |
| | *X-FSPMiner* | **743** | **624** | **614** | **551** | **499** | **385** | **352** | **390** | **350** | **339** |
| *Heart Disease* | *ObjectMiner* | 5463 | 1941 | 1117 | 679 | 513 | 385 | 330 | 279 | 250 | 225 |
| | *STreeDC-Miner* | **2467** | 1241 | 882 | 613 | 605 | 329 | 383 | 307 | 315 | 328 |
| | *X-FSPMiner* | 2543 | **1165** | **760** | **578** | **429** | **284** | **306** | **249** | **180** | **186** |
| *Indian Liver Patient* | *ObjectMiner* | 42855 | 32181 | 28101 | 24652 | 22204 | 19950 | 18151 | 16279 | 14813 | 13426 |
| | *STreeDC-Miner* | 17958 | 17437 | 15170 | 13819 | 12878 | 13505 | 13074 | 12133 | 11995 | 11087 |
| | *X-FSPMiner* | **8380** | **6694** | **5972** | **5547** | **5059** | **4481** | **4218** | **4006** | **3834** | **3131** |
| *Iris* | *ObjectMiner* | **93** | **77** | **65** | **58** | **54** | **49** | **46** | **43** | **42** | **41** |
| | *STreeDC-Miner* | 147 | 141 | 121 | 133 | 131 | 123 | 124 | 121 | 94 | 118 |
| | *X-FSPMiner* | 104 | 99 | 93 | 89 | 85 | 69 | 65 | 74 | 66 | 67 |
| *Metadata* | *ObjectMiner* | 3955160 | 4081705 | **20812** | **20643** | **10350** | **10327** | 7828 | 7852 | 7750 | 6518 |
| | *STreeDC-Miner* | **2631783** | **2480349** | 34449 | 33828 | 17647 | 17726 | 12216 | 12318 | 11682 | 9523 |
| | *X-FSPMiner* | 3518015 | 4005714 | 29263 | 29209 | 13777 | 13513 | **6793** | **6781** | **6376** | **4422** |
| *Poker Hand* | *ObjectMiner* | 142319 | 142200 | 136337 | 20451 | 20185 | 20285 | 20383 | 20526 | 20201 | 20770 |
| | *STreeDC-Miner* | 37795 | 33628 | 29391 | 13443 | 11859 | 10438 | 8960 | 9043 | 8682 | 8904 |
| | *X-FSPMiner* | **17638** | **16069** | **15341** | **4642** | **4568** | **4695** | **4706** | **4809** | **4563** | **4620** |
| *Teaching Assistant Evaluation* | *ObjectMiner* | **93** | 79 | **67** | **64** | **59** | **54** | **50** | **50** | **48** | **42** |
| | *STreeDC-Miner* | 138 | 107 | 125 | 132 | 99 | 127 | 101 | 94 | 118 | 119 |
| | *X-FSPMiner* | 102 | **76** | 88 | 84 | 64 | 77 | 76 | 59 | 69 | 67 |
| *Vehicle Silhouettes* | *ObjectMiner* | 263099 | 49099 | 20210 | 10916 | 6844 | 4586 | 3101 | 2350 | 1732 | 1349 |
| | *STreeDC-Miner* | 120438 | 30736 | 15141 | 10174 | 6593 | 5108 | 3699 | 3034 | 2130 | 2019 |
| | *X-FSPMiner* | **107173** | **28545** | **13560** | **8435** | **5859** | **4010** | **2399** | **1690** | **1282** | **998** |
| *Wine* | *ObjectMiner* | 2158 | 1442 | 1158 | 997 | 962 | 798 | 758 | 594 | 414 | 306 |
| | *STreeDC-Miner* | 789 | 452 | 366 | 342 | 300 | 307 | 303 | 267 | 277 | 199 |
| | *X-FSPMiner* | **671** | **404** | **320** | **229** | **255** | **238** | **229** | **164** | **190** | **147** |

The performance of algorithms in terms of the runtime is shown in Table 3. For each dataset and $minFreq$, the best runtime is marked in bold.

Although *X-FSPMiner* achieves the best runtime for all most cases, runtimes are not directly comparable for different combinations of $minFreq$ and $dataset$. However, for each $minFreq$ and $dataset$, how many standard deviations separate each algorithm runtime from the average runtime can be calculated. This measure, the standardized performance $sp$ of an algorithm $a$ for a $minFreq$ $b$ and a $dataset$ $c$, is defined as:

$$sp_{a,b,c} = \frac{avgRuntime_{b,c} - runtime_{a,b,c}}{stdRuntime_{b,c}} \tag{6}$$

where $runtime_{a,b,c}$ is the runtime of the algorithm $a$ for $minFreq$ $b$ a and dataset $c$, $avgRuntime_{b,c}$ is the average runtime of all algorithms for $minFreq$ $b$ and dataset $c$, and $stdRuntime_{b,c}$ is the standard deviation of all algorithms for $minFreq$ $b$ and dataset $c$. Notice that the more positive an algorithm's standardized performance is, the more efficient (relative to the others) the algorithm is.

The average of the standardized performance of an algorithm for a dataset overall explored $minFreqs$ summarizes its performance for the dataset consistently (see Figure 4). Analogously, the average of the standardized performance of an algorithm for a $minFreqs$ overall used datasets summarizes its performance for the $minFreqs$ (see Figure 5). Finally, a global average of all standardized performance of an algorithm summarizes its performance (see Figure 6).

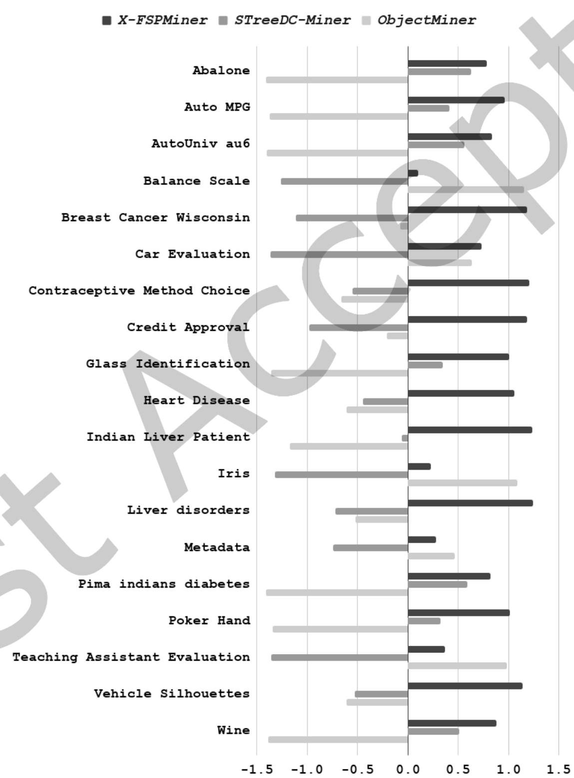


Fig. 4. Average standardized performance of *X-FSPMiner*, *STreeDC-Miner* and *ObjectMiner* by dataset.

In Figure 4, it can be seen that for all datasets, the behavior of *X-FSPMiner* is better than average behavior. Also, for most datasets, *X-FSPMiner* achieves the best average standardized performance. For only four datasets (Balance Scale, Iris, Metadata, Teaching Assistant Evaluation), the best average standardized performance was obtained by another algorithm (*ObjectMiner*).

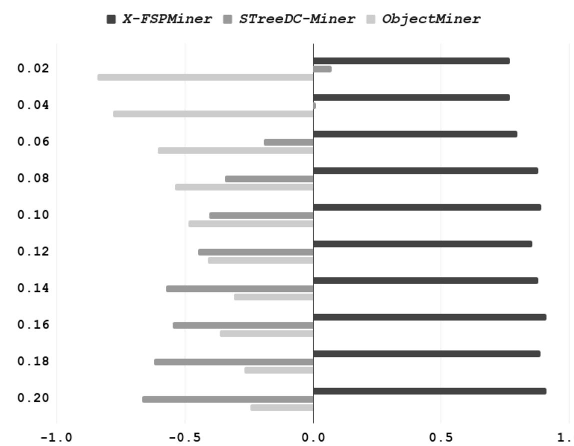


Fig. 5. Average standardized performance of *X-FSPMiner*, *STreeDC-Miner* and *ObjectMiner* by *minFreq*.

Figure 5 shows the average of the standardized performance of the algorithms for a *minFreqs* overall used datasets. From this point of view, it can be seen that the behavior of the *ObjectMiner* and *STreeDC-Miner* algorithms generally does stay behind the average performance. *ObjectMiner* shows a trend to improve its behavior when the *minFreqs* increases, while in contrast, *STreeDC-Miner* shows a trend to improve its behavior when the *minFreqs* decreases. However, *STreeDC-Miner*, unlike *ObjectMiner*, for the two smallest values of *minFreqs*, slightly outperforms average behavior. Differently, our proposed algorithm, *X-FSPMiner*, always outperforms the average behavior and is far superior to the other algorithms. Additionally, *X-FSPMiner* shows a slight tendency to improve its behavior with increasing *minFreqs*.

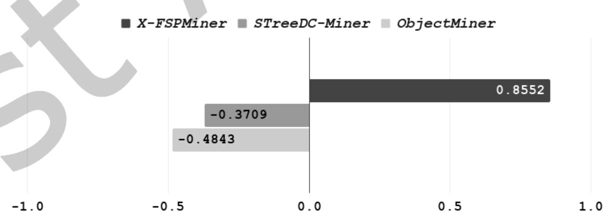


Fig. 6. Global average standardized performance of *X-FSPMiner*, *STreeDC-Miner* and *ObjectMiner*.

Finally, Figure 6 shows the global average of all standardized performance of each algorithm. It can be seen that both *ObjectMiner* and *STreeDC-Miner* have a general behavior below the average ($-0.37$ and $-0.48$ standard deviation from the average behavior, respectively). In contrast, the general behavior of *X-FSPMiner* is much higher (0.86 standard deviation from average behavior).

The results presented summarize the broad superiority of *X-FSPMiner* in terms of runtime over *ObjectMiner* and *STreeDC-Miner*.

## 6 CONCLUSIONS

In this paper, we proposed *X-FSPMiner*, an efficient algorithm for mine all FSP using Boolean and non-increasing monotonic similarity functions. *X-FSPMiner* employs the also proposed *FV-Tree* data structure to condense a collection of objects described by numerical and non-numerical features and to fast compute the frequency of the FSPs.

The results show that *X-FSPMiner* outperforms the average behavior of the state-of-the-art algorithms (*ObjectMiner* and *STreeDC-Miner*) in term of runtime. For most datasets tested, *X-FSPMiner* achieves the best average standardized performance. On the other hand, for all $minFreqs$ tested, the average standardized performance of *X-FSPMiner* is far superior to the other algorithms. Summarizing both points of view, the global average of all standardized performance of each tested algorithm reveals that the general behavior of our proposed algorithm, *X-FSPMiner*, is greater than the average behavior in 0.86 standard deviation. In comparison, the behavior of the other algorithms is lesser than the average behavior in −0.37 (*STreeDC-Miner*) and −0.48 (*ObjectMiner*) standard deviation.

Although our proposed algorithm, *X-FSPMiner*, vastly outperforms the state-of-the-art algorithms, some issues must be attended: I) There are real-world problems in which the study objects are compared using other types of similarity functions not supported by *X-FSPMiner* (e.g., non-boolean or increasing monotonic similarity functions). II) as the computation capabilities grow and information technologies are massively adopted, including the Internet of Things, the size of study object collections in terms of the number of objects and features also grows, implying that better computational performance is required. However, all the FSP mining algorithms (including *X-FSPMiner*) are designed for a single computation core. They do not take advantage of the current multiple CPU and GPU core capabilities. III) Although mining all FSP finds hidden knowledge, their main drawback is that many frequent similar patterns are mined, leading to expensive post-analysis and post-processing. As future work, we visualize addressing these issues as follows: I) Extend the *FV-Tree* data structure and *X-FSPMiner* algorithm for non-boolean similarity functions, and design a new FSP mining algorithm based on relaxed pruning for increasing monotonic similarity functions. II) Design a parallel FSP miner algorithm based on *X-FSPMiner* considering the capabilities of multicore CPU and GPU architectures. III) Developing a metaheuristic FSP miner algorithm for obtaining a representative subset of all FSP patterns.



## REFERENCES


[1] Charu C Aggarwal. 2014. Applications of frequent pattern mining. In *Frequent pattern mining*. Springer, 443–467.

[2] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. 1993. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*. 207–216.

[3] Rakesh Agrawal, Ramakrishnan Srikant, et al. 1994. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, Vol. 1215. 487–499.

[4] Nathalie Alemán-García and Martha R. Ortiz-Posadas. 2021. Evaluation of Hepatic Fibrosis Stages Using the Logical Combinatorial Approach. In *Progress in Artificial Intelligence and Pattern Recognition*, Yanio Hernández Heredia, Vladimir Milián Núñez, and José Ruiz Shulcloper (Eds.). Springer International Publishing, Cham, 158–166.

[5] Nader Aryabarzan, Behrouz Minaei-Bidgoli, and Mohammad Teshnehlab. 2018. negFIN: An efficient algorithm for fast mining frequent itemsets. *Expert Systems with Applications* 105 (2018), 129–143.

[6] Vyacheslav Busarov, Natalia Grafeeva, and Elena Mikhailova. 2016. A Comparative Analysis of Algorithms for Mining Frequent Itemsets. In *Databases and Information Systems*, Guntis Arnicans, Vineta Arnicane, Juris Borzovs, and Laila Niedrite (Eds.). Springer International Publishing, Cham, 136–150.

[7] Sung-Hyuk Cha. 2007. Comprehensive Survey on Distance/Similarity Measures between Probability Density Functions. *International Journal of Mathematical models and Methods in Applied Sciences* 1, 4 (2007), 300–307.

[8] Yangming Chen, Philippe Fournier-Viger, Farid Nouioua, and Youxi Wu. 2021. Sequence Prediction using Partially-Ordered Episode Rules. In *2021 International Conference on Data Mining Workshops (ICDMW)*. 574–580.

[9] Roxana Danger, José Ruíz-Shulcloper, and Rafael Berlanga Llavori. 2004. Objectminer: A New Approach for Mining Complex Objects. In *ICEIS (2)*. Citeseer, 42–47.

[10] ZhiHong Deng, ZhongHui Wang, and JiaJian Jiang. 2012. A new algorithm for fast mining frequent itemsets using N-lists. *Science China Information Sciences* 55, 9 (Sept. 2012), 2008–2030.

[11] Zhi-Hong Deng. 2014. Fast mining Top-Rank-k frequent patterns by using Node-lists. *Expert Systems with Applications* 41, 4, Part 2 (2014), 1763–1768.

[12] Zhi-Hong Deng. 2016. DiffNodesets: An efficient structure for fast mining frequent itemsets. *Applied Soft Computing* 41 (2016), 214–223.

[13] Zhi-Hong Deng and Sheng-Long Lv. 2015. PrePost+: An efficient N-lists-based algorithm for mining frequent itemsets via Children–Parent Equivalence pruning. *Expert Systems with Applications* 42, 13 (2015), 5424–5432.

[14] Philippe Fournier-Viger, Wensheng Gan, Youxi Wu, Mourad Nouioua, Wei Song, Tin Truong, and Hai Duong. 2022. Pattern Mining: Current Challenges and Opportunities. In *Database Systems for Advanced Applications. DASFAA 2022 International Workshops*, Uday Kiran Rage, Vikram Goyal, and P. Krishna Reddy (Eds.). Springer International Publishing, Cham, 34–49.

[15] J Gómez, O Rodríguez, S Valladares, J Ruiz-Shulcloper, et al. 1994. Prognostic of Gas-oil deposits in the Cuban ophiological association, applying mathematical modeling. *Geofisica Internacional* 33, 3 (1994), 447–467.

[16] G. Grahne and J. Zhu. 2005. Fast algorithms for frequent itemset mining using FP-trees. *IEEE Transactions on Knowledge and Data Engineering* 17, 10 (2005), 1347–1362.

[17] Jiawei Han, Hong Cheng, Dong Xin, and Xifeng Yan. 2007. Frequent pattern mining: current status and future directions. *Data Mining and Knowledge Discovery* 15, 1 (Aug. 2007), 55–86.

[18] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining Frequent Patterns without Candidate Generation. *SIGMOD Rec.* 29, 2 (May 2000), 1–12.

[19] Hieu Hanh Le, Tatsuhiro Yamada, Yuichi Honda, Takatoshi Sakamoto, Ryosuke Matsuo, Tomoyoshi Yamazaki, Kenji Araki, and Haruo Yokota. 2022. Methods for Analyzing Medical-Order Sequence Variants in Sequential Pattern Mining for Electronic Medical Record Systems. *ACM Trans. Comput. Healthcare* (sep 2022). Just Accepted.

[20] Carson Kai-Sang Leung. 2009. *Anti-monotone Constraints*. Springer US, Boston, MA, 98–98.

[21] Guimei Liu, Hongjun Lu, Wenwu Lou, Yabo Xu, and Jeffrey Xu Yu. 2004. Efficient Mining of Frequent Patterns Using Ascending Frequency Ordered Prefix-Tree. *Data Mining and Knowledge Discovery* 9, 2 (Nov. 2004), 249–274.

[22] Martha R. Ortiz-Posadas. 2017. The Logical Combinatorial Approach Applied to Pattern Recognition in Medicine. In *New Trends and Advanced Methods in Interdisciplinary Mathematical Sciences*, Bourama Toni (Ed.). Springer International Publishing, Cham, 169–188.

[23] Zhiwen Pan, Jiangtian Li, Yiqiang Chen, Jesus Pacheco, Lianjun Dai, and Jun Zhang. 2019. Knowledge discovery in sociological databases. *International Journal of Crowd Science* (2019).

[24] J. Pei, Jiawei Han, Hongjun Lu, S. Nishio, S. Tang, and Dongqing Yang. 2001. H-mine: hyper-structure mining of frequent patterns in large databases. *Proceedings 2001 IEEE International Conference on Data Mining* (2001), 441–448.

[25] Ansel Y Rodríguez-González, Fernando Lezama, Carlos A Iglesias-Alvarez, José Fco Martínez-Trinidad, Jesús A Carrasco-Ochoa, and Enrique Munoz de Cote. 2018. Closed frequent similar pattern mining: Reducing the number of frequent similar patterns without information loss. *Expert Systems with Applications* 96 (2018), 271–283.

[26] Ansel Y Rodríguez-González, José Francisco Martínez-Trinidad, Jesús Ariel Carrasco-Ochoa, and José Ruiz-Shulcloper. 2008. Mining frequent similar patterns on mixed data. In *Iberoamerican Congress on Pattern Recognition*. Springer, 136–144.

[27] Ansel Y Rodríguez-González, José Fco Martínez-Trinidad, Jesús Ariel Carrasco-Ochoa, and José Ruiz-Shulcloper. 2010. Using non boolean similarity functions for frequent similar pattern mining. In *Canadian Conference on Artificial Intelligence*. Springer, 374–378.

[28] Ansel Yoan Rodríguez-González, José Francisco Martínez-Trinidad, Jesús Ariel Carrasco-Ochoa, and José Ruiz-Shulcloper. 2011. RP-Miner: a relaxed prune algorithm for frequent similar pattern mining. *Knowledge and information systems* 27, 3 (2011), 451–471.

[29] Ansel Y. Rodríguez-González, José Fco. Martínez-Trinidad, Jesús A. Carrasco-Ochoa, and José Ruiz-Shulcloper. 2013. Mining frequent patterns and association rules using similarities. *Expert Systems with Applications* 40, 17 (2013), 6823–6836.

[30] Ansel Y Rodríguez-González, José F Martínez-Trinidad, Jesús A Carrasco-Ochoa, José Ruiz-Shulcloper, and Matías Alvarado-Mentado. 2019. Frequent similar pattern mining using non Boolean similarity functions. *Journal of Intelligent & Fuzzy Systems* 36, 5 (2019), 4931–4944.

[31] J Ruiz-Shulcloper and A Fuentes-Rodrguez. 1981. A cybernetic model to analyze juvenile delinquency. *Revista Ciencias Matemáticas* 2, 1 (1981), 123–153.

[32] Ashoka Savasere, Edward Omiecinski, and Shamkant B. Navathe. 1995. An Efficient Algorithm for Mining Association Rules in Large Databases. In *Proceedings of the 21th International Conference on Very Large Data Bases (VLDB '95)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 432–444.

[33] Pradeep Shenoy, Jayant R. Haritsa, S. Sudarshan, Gaurav Bhalotia, Mayank Bawa, and Devavrat Shah. 2000. Turbo-Charging Vertical Mining of Large Databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data* (Dallas, Texas, USA) *(SIGMOD '00)*. Association for Computing Machinery, New York, NY, USA, 22–33.

[34] Akbar Telikani and Asadollah Shahbahrami. 2018. Data sanitization in association rule mining: An analytical review. *Expert Systems with Applications* 96 (2018), 406 – 426.

[35] José Fco. Martínez Trinidad, José Ruiz Shulcloper, and Manuel S. Lazo Cortés. 2000. Structuralization of universes. *Fuzzy Sets and Systems* 112, 3 (2000), 485 – 500.
[36] Amos Tversky. 1977. Features of similarity. *Psychological review* 84, 4 (1977), 327.
[37] Bay Vo, Sang Pham, Tuong Le, and Zhi-Hong Deng. 2017. A novel approach for mining maximal frequent patterns. *Expert Systems with Applications* 73 (2017), 178–186.
[38] Michael Weisberg. 2012. Getting serious about similarity. *Philosophy of Science* 79, 5 (2012), 785–794.
[39] Y. . Woon, W. . Ng, and E. . Lim. 2004. A support-ordered trie for fast frequent itemset discovery. *IEEE Transactions on Knowledge and Data Engineering* 16, 7 (2004), 875–879.
[40] Jin Soung Yoo. 2019. Crime Data Warehousing and Crime Pattern Discovery. In *Proceedings of the Second International Conference on Data Science, E-Learning and Information Systems* (Dubai, United Arab Emirates) *(DATA '19)*. Association for Computing Machinery, New York, NY, USA, Article 40, 6 pages.
[41] M. J. Zaki. 2000. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering* 12, 3 (2000), 372–390.
[42] Mohammed J. Zaki and Karam Gouda. 2003. Fast Vertical Mining Using Diffsets. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Washington, D.C.) *(KDD '03)*. Association for Computing Machinery, New York, NY, USA, 326–335.
[43] Shuling Zhu. 2019. Research on data mining of education technical ability training for physical education students based on Apriori algorithm. *Cluster Computing* 22 (2019), 14811–14818.