A Recursive Algorithm for Mining Association Rules

Abdelkader Mokkadem1 · Mariane Pelletier1 · Louis Raimbault1,2

Received: 5 November 2021 / Accepted: 21 June 2022 © The Author(s), under exclusive licence to Springer Nature Singapore Pte Ltd 2022

Abstract
Mining frequent itemsets and association rules are an essential task within data mining and data analysis. In this paper, we introduce PrefRec, a recursive algorithm for finding frequent itemsets and association rules. Its main advantage is its recursiveness with respect to the items. It is particularly efficient for updating the mining process when new items are added to the database or when some are excluded. We present in a complete way the logic of the algorithm, and give some of its applications. After that, we carry out an experimental study on the effectiveness of PrefRec. We first compare the execution times with some very popular frequent itemset mining algorithms. Then, we do experiments to test the updating capabilities of our algorithm.

Keywords Association rule · Frequent itemset · Data mining · Recursive algorithm

Introduction
Frequent itemset and association rule mining are one of the fundamental problems in data mining and computational statistics. This data analysis method was first introduced by Agrawal et al. [3] for mining transaction databases. Even if this method was introduced in the context of Market Business Analysis, it has many applications in other fields, such as genetics, webmining, textmining. The problem can be stated as follows (Agrawal et al. [3]). Let $I = \{a_1, \ldots, a_n\}$ be a finite set of items. A transaction database is a set of transactions $T = \{t_1, \ldots, t_N\}$ where each transaction $t_i \subset I$, $1 \leq i \leq N$, represents a nonempty subset of items. An itemset $A$ is a subset of $I$; $A$ is a $k$-itemset if it contains $k$ items. The support of an itemset $A$ is denoted as $\text{supp}(A)$ and is defined as the number of transactions which contain $A$. The relative support of $A$ is $\text{freq}(A) = \text{supp}(A)/N$. $A$ is frequent if $\text{freq}(A) \geq \sigma$ where $\sigma$ is a user-specified minimum relative support threshold, called minSup. The problem of Frequent Itemset Mining (FIM) is to find all the frequent itemsets for any given minSup.

An association rule is an implication $A \Rightarrow B$ where $A$ and $B$ are two items. The support of a rule $A \Rightarrow B$ is defined as $\text{supp}(A \Rightarrow B) = \text{supp}(A \cup B)$, its relative support as $\text{freq}(A \Rightarrow B) = \text{supp}(A \Rightarrow B)/N$. The confidence of a rule $A \Rightarrow B$ is defined as $\text{conf}(A \Rightarrow B) = \text{supp}(A \Rightarrow B)/\text{supp}(A)$. The problem of Association Rule Mining (ARM) is to find all association rules having a relative support no less than minSup and a confidence no less than a user-defined threshold called minConf. Thus the problem must be solved in two steps:

Step 1. Frequent itemset mining (FIM) is to determine all the frequent itemsets, that is, all the itemsets having a relative support no less than minSup.

Step 2. Association rule mining (ARM) is to discover all the association rules satisfying the confidence condition by using the frequent itemsets found in Step 1.

It is well known that the first step is the key to discovering all the association rules. After determining the frequent itemsets in the first step, the solution to the second step is quite straightforward: just generate, for each frequent itemset $C$, all the rules $A \Rightarrow B$ where $A$ is a subset of $C$ and $B = C \setminus A$.

To carry out Step 1 (FIM), we have to organize all the itemsets. For that we use the order of the database $I$ from which we deduce the prefix tree; it is a tree whose nodes are
the itemsets and in which the father-child relation is a relation of inclusion. Most known algorithms use this structure.

The first efficient algorithm to frequent itemset mining is Apriori (Agrawal and Srikan [2], Agrawal et al. [4]). Since the Apriori algorithm was proposed, there have been extensive studies on the improvements or extensions of Apriori, e.g., hashing technique (Park et al. [13]), partitioning technique (Savasere et al. [15]), sampling approach (Toivonen [16]), dynamic itemset counting (Brin et al. [6]), incremental mining (Cheung et al. [8]), parallel and distributed mining (Agrawal and Shafer [11]; Cheung et al. [7]; Zaki et al. [19]), integrating mining with relational database systems (Sarawagi et al. [14]), information stored in bitMatrix (Huang et al. [16]), dynamic itemset counting (Brin et al. [6]), incremental technique (Savasere et al. [15]), sampling approach (Toivonen [16]). Among the most efficient algorithms that have been proposed, the most famous ones are Eclat (Zaki [18]), Fp-Growth (Han et al. [9]) and LCM (Uno et al. [17]). Both Apriori and Fp-growth methods mine frequent patterns from a set of transactions in horizontal data format, while Eclat algorithm explores the vertical data format and LCM uses both vertical and horizontal representation.

In this paper, we introduce PrefRec, a new algorithm for fast discovery of frequent itemsets and association rules mining. This algorithm is based on a construction of the prefix tree in successive stages. More precisely, for each k, we construct the prefix tree of the base \{a_1, \ldots, a_{k+1}\} using \{a_{k+1}\} and the prefix tree of the base \{a_1, \ldots, a_k\}. One main property of the algorithm is that it is recursive with respect to the items. This means that once the algorithm has treated the set of items I = \{a_1, \ldots, a_n\}, if a new item \{a_{n+1}\} is inserted into I, then updating the algorithm only uses the item \{a_{n+1}\} and the result obtained previously with I = \{a_1, \ldots, a_n\}. This property is particularly important in the case where the database is not fixed and the items arrive successively, k denoting the order of arrival of the item \{a_k\}. An example is that of a binary time series observed on N individuals at times \{1, \ldots, n\}. Other types of update are also straightforward, such as removing the first item, adding or removing a set of items.

We also carry out an experimental study. The first part of the experimental study is focused on the time performance of the frequent itemset mining algorithms and use several datasets. We consider PrefRec and some known algorithms. The first one is Apriori, the most used in many areas. The others are Eclat, Fp-Growth and LCM, and are among the most efficient algorithms. It emerges from this study that PrefRec supports the comparison very well and that it performs better in some cases. It also appears that PrefRec is particularly efficient in the case of time series. This is probably because it is recursive. In the second part of the experimental study, we carry out experiments concerning recursion (adding or removing items) that show the nice properties of PrefRec in this field.

For PrefRec, we use our own C++ implementation available at https://github.com/LouisRaimbault/PrefRec.

The rest of the paper is organized as follows. In “FIM, ARM and Trees”, we give some formalism and definitions to describe the dataset of transactions, the items and the itemsets. Then we present the recursive building of the tree of all itemsets and that of all frequent itemsets; we give the properties of these trees. In “The Recursive Algorithm, its Updating, and its Applications”, we present the two versions of our algorithm (the one for the FIM and the one for the ARM), their updating, and their applications. In “Experimental Study”, we give our experimental study and compare the performance of PrefRec with that of other algorithms.

**FIM, ARM and Trees**

**Usual Definitions in Data Mining**

Let I = \{a_1, a_2, \ldots, a_n\} be a finite set of items called the item base. A transaction database is a set of transactions \(T = \{t_1, t_2, \ldots, t_N\}\), where each transaction \(t_i \subseteq \{1, \ldots, n\}\), is nonempty. An itemset A is a subset of I; A may be the empty set \(\emptyset\). The itemset A is a k-itemset if it contains k items. The support of an itemset A is denoted as \(\text{supp}(A)\) and is defined as the number of transactions \(t_i\) such that \(A \subseteq t_i\). The relative support of A is \(\text{freq}(A) = \text{supp}(A)/N\); in the trivial case \(A = \emptyset\), \(\text{supp}(A) = N\) and \(\text{freq}(A) = 1\). We say that A is frequent if \(\text{freq}(A) \geq \sigma\) where \(\sigma\) is a user-specified minimum relative support threshold, called minSup.

A rule is an implication of the form \(A \Rightarrow B\) where A and B are itemsets such that \(A \cap B = \emptyset\); A is called the antecedent and B the consequent. The support of a rule \(A \Rightarrow B\) is defined as \(\text{supp}(A \Rightarrow B) = \text{supp}(A \cup B)\). The confidence of a rule \(A \Rightarrow B\) is defined as \(\text{conf}(A \Rightarrow B) = \text{supp}(A \Rightarrow B)/\text{supp}(A) = \text{freq}(A \cup B)/\text{freq}(A)\). The problem of mining association rules is to find all the association rules in a database having a relative support no less than \(\text{minSup}\) and a confidence no less than a user-defined minimum confidence threshold called \(\text{minConf}\).

Let C be an itemset; for each \(A \subseteq C\) we denote by \(R(A, C)\) the association rule \(A \Rightarrow C\). The support of \(R(A, C)\) is then \(\text{supp}(C)\) and the confidence \(\text{supp}(C)/\text{supp}(A)\). Given C, there are two trivial rules: \(R(C, C)\) whose confidence is 1, and \(R(\emptyset, C)\) whose confidence is \(\text{freq}(C)\). Note that the rule \(R(\emptyset, C)\) means “C is in the transaction”. Of course, these trivial rules are not to be considered. Let \(\sigma\) and \(\tau\) be the \(\text{minSup}\) and the \(\text{minConf}\), respectively. If \(C\) is a frequent itemset, i.e., an itemset such that \(\text{freq}(C) \geq \sigma\), then for each \(A \subseteq C\), A is frequent. Let
RULE(C, ρ) = \{ R(A, C), A \subseteq C, \text{conf}(R(A, C)) \geq ρ \} be the set of association rules of the form A ⇒ C \setminus A and satisfying the confidence condition. It is then easy to see that the set of all association rules satisfying the support and the confidence conditions is the union of the sets RULE(C, ρ) with C frequent.

**Formal Interpretation of the Itemsets**

The item base \( a_j, 1 \leq j \leq n \), is ordered by the index \( j \). Thus \( I = \{a_1, a_2, \ldots, a_n\} \) will also be written \( I = \{1, 2, \ldots, n\} \). Likewise, each itemset \( A = \{a_{j_1}, \ldots, a_{j_k}\} \) will also be represented by a vector \((j_1, j_2, \ldots, j_k)\), where the coordinates are increasing. The database \( T \) can, therefore, be represented as a sequence \((t_1, t_2, \ldots, t_N)\) where each \( t_i \) is a vector with increasing coordinates in \( \{1, 2, \ldots, n\} \).

Another way to represent \( T \) is as follows. To each item \( a_j, 1 \leq j \leq n \), we associate a random variable \( X^{(j)} = 1 \) if \( a_j \) is in the transaction and \( X^{(j)} = 0 \) if not. Each transaction \( t \) is thus seen as an observation of the random variable \((X^{(1)}, \ldots, X^{(n)})\). The database \( T = \{t_1, t_2, \ldots, t_N\} \) is then a sample \((X^{(1)}_i, \ldots, X^{(n)}_i)_{1 \leq i \leq N}\) of the random variable \((X^{(1)}, \ldots, X^{(n)}_i)\). Equivalently, the database \( T \) is the \( N \times n \) matrix whose line \( i \) is \( t_i = (X^{(1)}_i, \ldots, X^{(n)}_i) \); the column \( X^{(j)} \) represents the item \( a_j \).

To each \( k \)-itemset \( A = \{a_{j_1}, \ldots, a_{j_k}\} \), we associate the random variable \( Z^{(A)} = X^{(j_1)} \times X^{(j_2)} \times \cdots \times X^{(j_k)} \), which clearly takes the value 1 if and only if the transaction contains \( A \). We note that \((X^{(j_1)}_i \times X^{(j_2)}_i \times \cdots \times X^{(j_k)}_i)_{1 \leq i \leq N}\) are the observed values of \( Z^{(A)} \), so that \( \text{freq}(A) = \frac{1}{N} \sum_{i=1}^{N} x^{(j_1)}_i \times x^{(j_2)}_i \times \cdots \times x^{(j_k)}_i \).

In the particular case \( A = \emptyset \), \( Z^{(A)} \) is the constant variable whose unique value is 1, and \( \text{freq}(A) = 1 \).

**Tree Enumeration of the Itemsets**

The subset relationships between itemsets form a partial order on the set itemsets. The most complete way to describe this set is the Hasse diagram; it is the graph where the nodes are the itemsets and where there is an edge between the two itemsets \( I, J \) if and only if \( J = I \cup \{a\} \) where \( a \) is an item. However, this representation leads to redundant search because the same itemset can be built multiple times by adding its items in different orders. To eliminate this redundancy, the Hasse graph is usually reduced to a tree called prefix tree.

Let us briefly recall some elements on trees. We will only consider planar trees and thus we define a tree as a planar-connected graph without cycle. We always assume the tree has a root, i.e. a distinguished node. Figure 1 gives a tree \( T \) with 16 nodes and a root \( R \).

For each node \( m \), there is a unique path between \( m \) and the root; the depth of the node \( m \) is the length of this path. The adjacent node to \( m \) in this path is the father of \( m \), the other adjacent nodes are the children of \( m \). In Fig. 2, \( f \) is the father of \( m \). The children of \( m \) are the nodes 1, 2, 3, 4; they are ordered in the clockwise order. We will say that 1 is the most left (with respect to \( m \)) and 4 is the most right. Of course, when \( m \) is the root, we must specify who is the first child of \( m \) in the clockwise order (or, equivalently, the most left).

The way we will visit the tree is the Depth First Search (DFS). There are two DFS, the Left DFS (LDFS) and the Right DFS (RDFS). They start from the root. From a node \( m \), the LDFS goes to the most left non-visited child of \( m \) if it exists and otherwise returns to the father of \( m \), while the RDFS goes to the most right non-visited child of \( m \) if it exists. Figures 3 and 4 give the two DFS for the tree \( T \) in Fig. 1.

**Prefix Tree**

The tree enumeration of the itemsets is based on the order of the items. Let \( I = \{1, 2, \ldots, n\} \) be the ordered set of the items; a \( k \)-itemset is represented by a vector with \( k \) increasing integers coordinates in \( \{1, 2, \ldots, n\} \). We define the prefix

---

Fig. 1  The tree \( T \) with 16 nodes and a root \( R \)

Fig. 2  Father, children, and the clockwise order
tree, \( pT_n \), a tree whose nodes are the itemsets, as follows. We proceed by defining successively each generation. This ensures that we get a planar tree.

- The root of \( pT_n \) is the empty itemset (\( \emptyset \)).
- The children of (\( \emptyset \)) are the 1-itemsets \( (1), (2), \ldots, (n) \) respecting their order and such that \( (1) \) is the most right (with respect to the root) and \( (n) \) is the most left.
- The children of the node \( A = (i_1, \ldots, i_k) \) are the nodes \( (i_1, \ldots, i_k, j) \) where \( i_k < j \leq n \) and \( j \) is ordered from the right (of \( A \)) to the left. More precisely, \( (i_1, \ldots, i_k, i_k + 1) \) is the most right child of \( A \) and \( (i_1, \ldots, i_k, n) \) is the most left. Of course if \( i_k = n \), then \( A \) is a leaf of the tree.

Figure 5 gives \( pT_4 \).

**Proposition 1** For a node \( a = (i_1, \ldots, i_k) \) in \( pT_n \), \( h(a) = k \) is the depth of \( a \).

**Proof** The nodes \( (1), (2), \ldots, (n) \) are the children of the root (\( \emptyset \)) and thus their depth is equal to 1. Assume the proposition true for \( k - 1 \geq 1 \), and let \( a = (i_1, \ldots, i_k) \); \( a \) is a child of the node \( r = (i_1, \ldots, i_{k-1}) \) and thus \( h(a) = h(r) + 1 \). Since \( h(r) = k - 1 \) by assumption, the result follows. \( \square \)

From the definition of the prefix tree, it is easy to deduce the following proposition.

**Proposition 2** Let \( pT_n \) be the prefix tree for the set of items \( I = \{1, 2, \ldots, n\} \). The following assertions are equivalent:

(i) \( a = (i_1, \ldots, i_k) \) is a leaf,
(ii) \( i_k = n \),
(iii) \( a = (i_1, \ldots, i_k) \) is the most left child of \( (i_1, \ldots, i_{k-1}) \).

We have the following obvious property.

**Property 1** Let \( m = (i_1, \ldots, i_k) \) be an itemset and \( pT_n(m) \) be the subtree rooted in the node \( m \) in \( pT_n \). The set of the nodes of \( pT_n(m) \) is composed of \( m \) and of all the nodes...
Let \( A \subset B \) then \( \{ \text{freq}(A) < \sigma \Rightarrow \text{freq}(B) < \sigma \} \). \hspace{1cm} (1)

In other words, if \( B \) is a frequent itemset, then all the itemsets included in \( B \) are frequent.

Since \( \text{freq}(\emptyset) = 1 \), \( \emptyset \) is always frequent. In view of (1) and of Property 1, if a node \( m \) of \( pT_n \) is not frequent, then none of the nodes in the subtree rooted in the node \( m \) is frequent. Consequently, by removing all the nodes \( a \) such that \( \text{freq}(a) < \sigma \) in \( pT_n \), we obtain a tree with \( \emptyset \) as root. We can thus define the frequent prefix tree as follows.

**Definition 1** Let \( \sigma \) be the minSup. We call frequent prefix tree, the tree \( pT_n^\sigma \) obtained from \( pT_n \) by removing the nodes \( a \) such that \( \text{freq}(a) < \sigma \).

**Proposition 3** If \( pT_n^\sigma(m) \) is the subtree rooted in the node \( m \) in the frequent prefix tree \( pT_n^\sigma \), then \( m \) is frequent and \( pT_n^\sigma(m) \) is obtained by removing from \( pT_n^\sigma(m) \) all the nodes \( a \) such that \( \text{freq}(a) < \sigma \).

**Remark 1** If \( m \) is not frequent, then \( m \) is not in \( pT_n^\sigma \) and \( pT_n^\sigma(m) \) is empty.

**Remark 2** If \( m \) is not in \( pT_n^\sigma \), then none of the nodes of \( pT_n(m) \) are in \( pT_n^\sigma \).

**Remark 3** By Property 1, we have \( m \subset a \) for each node \( a \) of \( pT_n^\sigma(m) \).

---

**Fig. 6** Recursive prefix tree

\( k = 1, 2, 3 \) and their LDFS

\[ pT_1 = \emptyset + (1) \]

\[ \text{LDFS}(1) = (\emptyset, (1)) \]

\[ pT_2 = pT_1 + (2) \]

\[ \text{LDFS}(2) = (\emptyset, (2), (1, 2)) \]

\[ pT_3 = pT_2 + (3) \]

\[ \text{LDFS}(3) = (\emptyset, (3), (2), (2, 3), (1), (1, 3), (1, 2), (1, 2, 3)) \]

---

**Recursive Building of the Prefix Trees**

**The Recurrence Relation Between the Prefix Trees**

Let \( a = (i_1, \ldots, i_k) \) be an itemset, let \( j \) be an item such that \( j > i_k \); we set \( (a, j) = (i_1, \ldots, i_k, j) \). In the particular case \( a = (\emptyset) \), we set \( ((\emptyset), j) = (j) \).

Let \( pT_0 \) be the tree with one node \( \emptyset \). We recall that \( pT_k \) is the prefix tree based on the items \( \{1, \ldots, k\} \). As an example, \( pT_3 \) is a tree with two nodes, \( \{\emptyset, (1)\} \), where \( \emptyset \) is the root and \( (1) \) is its child.

**Proposition 4** The leaves of \( pT_{k+1} \) are the nodes \( (a, k+1) \) where \( a \) is a node of \( pT_k \). The tree \( pT_k^\sigma \) is obtained from \( pT_{k+1} \) by removing all the leaves \( (a, k+1) \).

**Proof** From Proposition 2, the set of leaves of \( pT_{k+1} \) is exactly the set of nodes \( (a, k+1) \) where \( a \) is an itemset based on \( 1, \ldots, k \), that is, a node of \( pT_k \). The edge relation between such itemsets \( a \) is the same in \( pT_k \) and in \( pT_{k+1} \). The result follows.

As a consequence, we have the following corollary, which gives the way to build the tree \( pT_n \) recursively (as an illustration, see Fig. 6).

**Corollary 1** The tree \( pT_{k+1} \) is built from \( pT_k \) by adding \( (a, k+1) \) as the most left child of \( a \) for each node \( a \) in \( pT_k \).

We can now state the following proposition, which gives the way to build the frequent tree \( pT_n^\sigma \) recursively.

**Proposition 5** (Recursion) Let \( \sigma \) be the minSup. The tree \( pT_{k+1}^\sigma \) is built from \( pT_k^\sigma \) as follows. For each node \( a \) in \( pT_k^\sigma \), if \( (a, k+1) \) is frequent, then it is added as the most left child of \( a \).
Proof $pT_{k+1}$ is obtained from $pT_k$ by adding the nodes $(a, k+1)$ with $a$ in $pT_k$. $pT_{k+1}$ is obtained from $pT_{k+1}$ by removing all the non frequent nodes. Equivalently, to obtain $pT_{k+1}$ we can first remove all the non frequent nodes of $pT_k$ and then add only the frequent nodes $(a, k + 1)$.

Ordering the Nodes of $pT_k$ and $pT_k^g$

For the construction of $pT_{k+1}$ we need to visit the nodes $a$ of $pT_k$ and to test if $(a, k+1)$ is frequent. In fact, due to the following proposition, we do not need to visit all the nodes $a$.

Proposition 6 (Pruning) Let $\sigma$ be the minSup and let $m$ be a node of $pT_k$. If $\text{freq}((m, k + 1)) < \sigma$, then freq($((b, k + 1)) < \sigma$ for all the nodes $b$ in the subtree $pT_k^g(m)$.

Proof In view of Remark 3, $m \in b$. Thus $(m, k + 1) \subset (b, k + 1)$ and $\text{freq}((b, k + 1)) \leq \text{freq}((m, k + 1))$. □

Remark 4 Let $m$ be a node in $pT_k$. If $m$ is in $pT_k^g$ and $(m, k + 1)$ is not in $pT_{k+1}^g$, then, for any node $a$ in $pT_k(m)$, $(a, k + 1)$ is not in $pT_{k+1}^g$.

In view of Proposition 6, we must visit the nodes of $pT_k^g$ in such a way that each node $m$ is visited before all the other nodes of its subtree $pT_k^g(m)$. Among the most well-known tree-traversing that satisfy the above, there are the Breadth-First Search (BDS) and the Depth First Search (DFS). For reasons of efficiency in the implementation of our algorithm, we choose to use the Depth First Search traversal. Moreover, to keep our algorithm recursive, we need to simultaneously carry out the search of the frequent itemsets and the search of the association rules. We are then led to use the Left Depth First Search traversal (LDFS).

Proposition 7 Let $r$ be a node in $pT_k$ (respectively, in $pT_k^g$). If a node $m$ is such that $m \in r$, then $m$ is before $r$ in $pT_k$ (respectively, in $pT_k^g$), with respect to the LDFS order. In particular $m$ is in $pT_k$ (respectively, in $pT_k^g$).

Proof We first prove Proposition 7 by induction in the case $r$ is in $pT_k$. The property is obvious for $pT_1$. From the construction of $pT_{k+1}$, we have the following two properties:

(i) for any $m$ in $pT_k$, $(m, k + 1)$ comes after $m$ in $pT_{k+1}$.
(ii) if $m$ is before $r$ in $pT_k$, then $(m, k + 1)$ is before $(r, k + 1)$ in $pT_{k+1}$.

Assume Proposition 7 is true for $pT_k$. Let $r$ in $pT_{k+1}$ and not in $pT_k$. Then $r = (v, k + 1)$ with $v$ in $pT_k$. Let a node $m$ be such that $m \subset r$. Either $m$ is a subset of $v$ and thus it is before $v$ (by induction assumption), which is before $r$ by property (i). Or $m = (w, k + 1)$ where $w$ is a subset of $v$ and thus $w$ is before $v$ in $pT_k$ (by induction assumption); it follows from (ii) that $m$ is before $r$ in $pT_{k+1}$.

Let us now consider the case $r$ is in $pT_k^g$. Then, $r$ is in $pT_k$ and thus $m$ is before $r$ in $pT_k$. Since $r$ is frequent and since $m \subset r$, $m$ is frequent and thus $m$ is before $r$ in $pT_k^g$.

Remark 5 Proposition 7 is the key to ensure that we can simultaneously carry out the mining of the frequent itemsets and the association rules. Indeed, it says that the order is such that, when a frequent itemset $C$ is found, all the $A \subset C$ have been tested before $C$ and their relative support have been computed.

Remark 6 Clearly Proposition 7 does not hold when the nodes in $pT_k$ are ordered with respect to the RDFS order. For example, the node $(1, 2, 3)$ is then before its subsets $((1, 3), (2), (2, 3), (3))$ in $pT_k$.

Before starting the presentation of the algorithm we need the following definition.

Definition 2 Let $a = (j_1, \ldots, j_{i-1}, j_i)$ be a node. If $j_i > j_{i-1} + 1$, we set $H(a) = \{b = (j_1, \ldots, j_{i-1}, s, j_i), j_{i-1} < s < j_i\}$. If $j_i = j_{i-1} + 1$, $H(a) = \emptyset$.

We note that if $a$ is not frequent, all the nodes in $H(a)$ are not frequent. Note also that the nodes in $H(a)$ are children of brothers of $a$.

The Recursive Algorithm, its Updating, and its Applications

We first present both versions of our algorithm: the first one, the PrefRec FIM algorithm, only gives the frequent itemsets, whereas the second one, the PrefRec ARM algorithm, also gives the association rules. Then, we explain how the algorithm PrefRec can be updated, and finally give its applications.

The PrefRec FIM Algorithm

The nodes $a$ of a tree $T$ are represented by a vector with increasing integer coordinates $a = (j_1, \ldots, j_i)$. To $a$ is associated the observation vector

$$(c^a_i)_{i \in \mathbb{SN}} = (c^{(j_1)}_i \times \cdots \times c^{(j_l)}_i)_{i \in \mathbb{SN}} \in \{0, 1\}^N$$
and its support \( \text{supp}(a) = \sum_{1 \leq i \leq N} \xi_i. \) The root of \( T \) is always the empty vector \((\emptyset)\); it is associated to \( \mathbf{1}_N = (1, 1, \ldots, 1) \in \{0, 1\}^N \) and \( \text{supp}(\emptyset) = N. \) The tree \( T \) is visited with the help of the LDFS; this gives a total order on the set of nodes. The PrefRec FIM algorithm is based on Proposition 5 and Proposition 6. For each \( k, 1 \leq k \leq n, \) this algorithm gives the tree \( pT_k \) whose nodes are the frequent itemsets based on \( \geq \kappa \) as child of \( z = (k \subset \leq n) \). The algorithm consists of successive applications of the FIM-Traversal function.

**PrefRec FIM Algorithm**

1. \( M = \{(k), 1 \leq k \leq n\} \) is the set of given 1-items and each \( (k) \) is associated to the observation vector \((x_i^{(k)})_{1 \leq i \leq N}\)
2. \( \sigma \) is the minSup
3. \( T_0 = \text{tree with one node} = \{(\emptyset)\} \)
4. For \( k = 1 \) to \( n \) do \( T_k = \text{FIM-Traversal}(T_{k-1}, (k), \sigma) \)

**FIM-Traversal function** The FIM-Traversal function takes as arguments:

- \( \sigma \) the minSup
- \( (k) \in M \)
- A tree \( T \) with root \((\emptyset)\) and whose nodes are itemsets \( a = (j_1, \ldots, j_l), j_f < k, \) associated to the observation vectors \((x_i^{(a)})_{1 \leq i \leq N}. \) The support of each node \( a \) is given and satisfies \( \text{supp}(a) \geq \sigma N. \)
- To each node \( a \) of \( T \) is associated a value \( v(a) \in \{0, 1\}. \)

The function returns a tree \( T' \) with its LDFS and whose set of nodes contains the set of nodes of \( T \) with order preserving. The function FIM-Traversal\((T, (k), \sigma)\) works as follows:

1. Initialize \( v(a) = 1 \) for all nodes \( a \) of \( T \)
2. Traverse the nodes of \( T \) in the LDFS order
3. While the current node \( a \) has not been visited before, do if \( v(a) = 1 \) then compute \( \text{supp}((a, k)) = \sum_{1 \leq i \leq N} (x_i^{(a)} \times x_i^{(k)}) \) if \( \text{supp}((a, k)) \geq \sigma N \) then add to \( T \) the node \((a, k)\) as the most left child of \( a, \) save \( \text{supp}((a, k)) \), and go to the next node after \( a \)
else For all \( m \) in \( H(a) \) set \( v(m) = 0 \) and go to the first node \( b \) such that \( b \not\in T(a) \) where \( T(a) \) is the subtree of \( T \) rooted in the node \( a \)
end if
else go to the first node \( b \) such that \( b \not\in T(a) \) where \( T(a) \) is the subtree of \( T \) rooted in the node \( a \)
end if

4. At the end of this function, the obtained tree is \( \text{FIM-Traversal}(T, (k), \sigma). \)

The heart of the algorithm is the FIM-Traversal function; it is the key of the recursiveness. When the set \( M \) of items is not given, but the items come successively, the algorithm PrefRec starts with an initial tree (e.g. \((\emptyset)\)) of frequent itemsets and then applies the FIM-Traversal function to each new item \((k)\).

**The PrefRec ARM Algorithm**

**The RULE function**

We first define the function RULE. This function takes as arguments a frequent itemset \( C \) and a minConf \( \tau \) and returns the set of association rules \( \text{RULE}(C, \tau) = \{R(A, C), A \subset C, \text{conf}(R(A, C)) \geq \tau\}. \)

More precisely, the function returns a tree \( T_C \) whose nodes are the itemsets \( A \subset C \) such that \( \text{conf}(R(A, C)) = \text{supp}(C)/\text{supp}(A) \geq \tau. \) The building of \( T_C \) uses the following pruning property

\[ B \subset A \subset C \text{ and } \text{conf}(R(A, C)) < \tau \Rightarrow \text{conf}(R(B, C)) < \tau. \]

Of course the function gives also the values \( \text{conf}(R(A, C)) \) for each node \( A \) of \( T_C. \) We use the RDFS but we may also use the LDFS.

**The RULE function**

1. \( T_c = \text{tree with one node} = \{C\} \) and \( C = (i_1, \ldots, i_m) \)
2. \textbf{Step 1} For each \( A \subset C \) with size \( m-1, \) if \( \text{supp}(C)/\text{supp}(A) \geq \tau, \) add \( A \) in \( T_C \) as a child of \( C. \)
3. \textbf{Step 2} Traverse the nodes of \( T_C \) in the RDFS order while the current node \( a \) has not been visited before, do if \( a \) is 1-itemset then go to the next node after \( a \) in the RDFS of \( T_C \)
else add in \( T_C \) as child of \( a \) all the \( u = a \cap b \) where \( b \) is brother of \( a \) in the left of \( a \) satisfying \( \text{supp}(C)/\text{supp}(a \cap b) \geq \tau \) and go to the next node after \( a \) in the RDFS of \( T_C \)
end if
end while
4. At the end of the RULE function, the obtained tree is \( T_C = \text{RULE}(C, \tau). \)

**The PrefRec ARM Algorithm**

For each \( k, 1 \leq k \leq n, \) the PrefRec ARM algorithm gives the tree \( pT_k \) whose nodes are the frequent itemsets based on
the items \{1, \ldots, k\}, as well as the set \text{RULE}(C, \tau) for each node \(C\) in \(pT^\sigma_{k}\).

\textbf{PrefRec ARM Algorithm} In PrefRec FIM algorithm, replace the FIM-Traversal function by the ARM-Traversal function.

\textbf{The ARM-Traversal function} The ARM-Traversal function takes as arguments those of the FIM-Traversal function increased by the \(\minConf\) \(\tau\).

The function returns a tree \(T'\) with its LDFS and whose set of nodes contains the set of nodes of \(T\) with order preserving and gives, for each new node \(A\), the set \(\text{RULE}(A, \tau)\).

The function ARM-Traversal\((T, (k), \sigma, \tau)\) works as the FIM-Traversal function where the statements

\begin{itemize}
  \item \textbf{if} \(\text{supp}((a, k)) \geq \sigma N\) \textbf{then} add to \(T\) the node \((a, k)\) as the most left child of \(a\), save \(\text{supp}((a, k))\), and go to the next node after \(a\)
\end{itemize}

are replaced by

\begin{itemize}
  \item \textbf{if} \(\text{supp}((a, k)) \geq \sigma N\) \textbf{then} add to \(T\) the node \((a, k)\) as the most left child of \(a\), save \(\text{supp}((a, k))\), compute \(\text{RULE}((a, k), \tau)\), and go to the next node after \(a\)
\end{itemize}

Of course, when the database \(M = \{(k), 1 \leq k \leq n\}\) is fixed, instead of applying PrefRec ARM Algorithm, we can first apply PrefRec FIM Algorithm and then apply the Rule function to all the frequent itemsets found.

\textbf{Updating}

At each step \(n\), we can introduce a new item and/or remove the first item; we consider both operations in the FIM case, the ARM case being easily deduced. The \(\minSup\) value is still \(\sigma\).

\textbf{Adding a New Item}

After doing step \((n - 1)\), we got the tree of frequent itemsets \(pT^\sigma_{n-1}\) for the set of items \(I = \{1, \ldots, n - 1\}\). When the item \((n)\) is introduced, the tree \(pT^\sigma_{n}\) of all the frequent itemsets for the set of items \(I = \{1, \ldots, n - 1, n\}\) is obtained only with the use of \((n)\) and \(pT^\sigma_{n-1}\) (see Proposition 5). More precisely, this step \((n)\) is the application of the FIM-Traversal function to \((n)\) and \(pT^\sigma_{n-1}\). Of course, for the association rule mining, the updating is the application of the ARM-Traversal function.

\textbf{Removing the First Item}

Suppose that in the current step the set of items is \(I = \{r, \ldots, n\}\). Let \(pT^\sigma_{r,n}\) be the frequent prefix tree based on \(I\). Moreover, assume that \((r)\) is frequent, so that \((r)\) is in \(pT^\sigma_{r,n}\). We want to remove \((r)\) and all the frequent itemsets containing \((r)\). From the construction of \(pT^\sigma_{r,n}\), we see that \((r)\) is the most right child of the root \((\emptyset)\) and all the frequent itemsets containing \((r)\) are in the subtree rooted in the node \((r)\). We have just to remove this subtree in \(pT^\sigma_{r,n}\). Of course for each deleted itemset \(C\), all the rules of type \(R(A, C)\) are also deleted.

\textbf{Application: Moving FIM and Moving ARM}

In many applications, it is interesting to have the successive trees of the frequent itemsets based on a fixed number \(q\) of frequent items. The Moving FIM algorithm (MFIM) responds to this concern. The items \((i)\) of the database arrive as they go. In a first step, using the new item if it is frequent, the algorithm builds the tree of the frequent itemsets based on the first \(q\) frequent items. In the second step, if the new item is frequent, the algorithm performs two tasks. In the first task it deletes all the frequent itemsets that contain the most right child of \((\emptyset)\), that is, the 1-itemset which has spent the most time in the tree. In the second task it adds the frequent itemsets that contain the new frequent item. The algorithm stops once it has performed \(Q\) frequent item replacements. In the algorithm, the index \(i\) lists the items which successively enter the database and the index \(k\) lists those which are frequent.

Another interpretation of the Moving FIM application is to consider only the set of \(q + Q\) frequent items \(\{1, \ldots, q + Q\}\), and to see it as a moving basis of \(q\) frequent items: for each integer \(k, 1 \leq k \leq Q + 1\), we want to find the set \(T^\sigma_{k,q}\) of frequent itemsets based on the set of \(q\) items \(\{k, \ldots, k + q - 1\}\).

For any planar tree \(T\) with root \((\emptyset)\), we denote by \(\text{DEL}(T)\) the tree obtained from \(T\) by deleting the most right child of \((\emptyset)\) and its subtree.

\textbf{MFIM Algorithm}

1. \(k = 0, T^\sigma_{0,q} = \{(\emptyset)\}, i = 0\)
2. \textbf{while} \(k \leq q, \textbf{do}\)
   \(i = i + 1\)
   \textbf{if} \((i)\) is frequent, \textbf{then}
   \(k = k + 1\)
   \(T^\sigma_{k,q} = \text{FIM-Traversal}(T^\sigma_{k-1,q}, (k), \sigma)\)
\textbf{end if}
\textbf{end while}
3. \textbf{while} \(k \leq q + Q, \textbf{do}\)
   \(i = i + 1\)
   \textbf{if} \((i)\) is frequent, \textbf{then}
   \(k = k + 1\)
   \(T = \text{DEL}(T^\sigma_{k-1,q})\)
   \(T^\sigma_{k,q} = \text{FIM-Traversal}(T, (k), \sigma)\)
\textbf{end if}
\textbf{end while}
If, for each integer \( k \), we want to find \( T^q_{k,q} \) and, for each node \( C \) of \( T^q_{k,q} \), the set \( R(C, \tau) \), we can use the following Moving ARM algorithm (MARM).

**MARM Algorithm**  In **MFIM Algorithm**, replace the FIM-Traversal function by the ARM-Traversal function.

### Experimental Study

We conducted an experimental study on the recursion properties of PrefRec and compared it to other algorithms on computation time. For our experiments, we chose Eclat, LCM, Fp-Growth and Apriori which are among the most used and efficient algorithms for finding frequent itemsets (see Borgelt [5]).

For these algorithms, we have used the implementations of Borgelt which can be found in [https://borgelt.net/software.html](https://borgelt.net/software.html). Note that Borgelt’s implementation of Eclat and LCM is in fact the implementation of an optimized mixture of Eclat and LCM, denoted Eclat/LCM. There are other older implementations of these four algorithms that can be found in Goethals’ webpage [http://adrem.uantwerpen.be/~goethals/software/index.old.html](http://adrem.uantwerpen.be/~goethals/software/index.old.html), as well as in [http://fimi.uantwerpen.be/src/](http://fimi.uantwerpen.be/src/). It turns out that these older implementations of Apriori and Fp-Growth are less efficient than the implementation of Borgelt that we have chosen. The older implementation of Eclat is very less efficient than Borgelt’s Eclat/LCM mixture while that of LCM is quite equivalent. We, therefore, have chosen to use Eclat/LCM. The mixture Eclat/LCM gives the possibility to select certain parameters which can affect the runtime. For all the datasets processed, we left the auto-selection of parameters which, in our case, turned out to choose the best function for almost all cases.

To run PrefRec, we have developed our own C++ implementation available at [https://github.com/LouisRaimbault/PrefRec](https://github.com/LouisRaimbault/PrefRec). The compilation of our functions is performed with the O3 parameters of the gcc compiler. It also includes some usual pragma directives and therefore the effectiveness may vary by machine, mainly using function to calculate the number of 1s in a bitfield. The experiments are realized on “Ubuntu 20.01 1 LTS” with characteristics: Intel Core i7-8650U CPU 1.95GHz * 8, operating system 64 bits, RAM capacity 16 GB (2*8), RAM speed 2400 Mhz.

Our first objective is to compare the four algorithms on the computation time necessary for the extraction of frequent itemsets, which corresponds to almost the entire time of extraction of the association rules. This study, presented in “Comparison with Non-recursive Algorithms”, does not concern recursion properties: the databases used are fixed. Let us mention that the usual known non-recursive algorithms, to reduce the computation time, do some actions on the database before launching the algorithm: the non-frequent items are deleted, and the others are reordered according to their support. In this comparative study, the item base is fixed and we do not consider the recursion properties. So with PrefRec also, we first remove the non-frequent items but the algorithm is applied without reordering the database. However, this preprocessing of the database is not taken into account in the computation time.

Our second purpose is to present in “Applications of Recursion Properties”, some recursive updating applications with PrefRec. We will present the application which consists in adding new variables to the database, as well as the Moving FIM application. Of course, in this experimental part, since the database is not fixed, we cannot preprocess it. We need to test the items as they arrive.

Among the datasets we consider, Chess, Accident, Pumsb and T40I10D100K are taken from [http://fimi.uantwerpen.be/data/](http://fimi.uantwerpen.be/data/). All the others are available at [https://github.com/LouisRaimbault/PrefRec](https://github.com/LouisRaimbault/PrefRec).

### Comparison with Non-recursive Algorithms

The aim of this section is to compare PrefRec with the three algorithms Eclat/LCM, Fp-Growth and Apriori. These algorithms being non-recursive, they must have the entire database before being launched. The applications presented in this section are therefore non-recursive and are carried out with fixed databases, that is, databases whose set of 1-items does not change during the process.

We compare the four algorithms on the computation time. The execution time of an algorithm depends on several parameters among which we can quote: the number of transactions \( N \), the number of items \( n \), the supports of the items, the transaction sizes, the minSup, the number of frequent items, the number of frequent itemsets.

We compare the execution times according to the minSup. We used 10 datasets, each with 7 values of minSup.

In the first part, we consider four well-known datasets in data mining, namely Chess, Accident, Pumsb and T40I10D100K. These four datasets have different characteristics. The number of items and the number of transactions are respectively 75 and 3196 for Chess, 468 and 340183 for Accident, 2113 and 49046 for Pumsb, 942 and 100000 for T40I10D100K.

In the second part we consider six synthetic datasets; two are built with independent variables and the four others are built with time series. For these six datasets, the number of transactions \( N \) is 100000. For the number of items \( n \), we used the values 1000, 4000 and 6000.

It turns out that the results and conclusions of both parts are different.

On each graph, for a sake of representation, we do not present points of an algorithms if its runtime values are too slow compared to the others.
In Table 1 we give, for the four datasets Chess, Accident, Pumsb and T40I10D100K, the number of frequent items (line called items) and the number of frequent itemsets (line with the dataset name) according to the minSup values. In Fig. 7 we present the execution times to make these extractions. On the x-axis it is minSup and on the y-axis it is time in seconds.

In view of the results, we first notice that Apriori has many difficulties and is clearly very inefficient compared to the other algorithms. It is clear that Fp-Growth performs better than all the others for the Chess, Accident and Pumbs datasets. Eclat/LCM performs better than PrefRec for Chess and Pumbs. But it does not do much better for the Accident dataset where it seems to lose some effectiveness. This is probably due to the fact that the number of transactions N is large in the Accident database.

The results are different with the T40I10D100K dataset. In this case, Eclat/LCM and PrefRec are the best with a slight advantage for PrefRec. Both algorithms work slightly better than Fp-Growth which seems to have some difficulties with the T40I10D100K dataset.

The main difference between T40I10D100K and the other datasets is the number of frequent 1-items (see Table 1). For T40I10D100K there are more than 860 frequent 1-items, while for the others the number of frequent 1-items is less than 76. It, therefore, seems that Fp-Growth is suitable for the case where the number of frequent 1-items is small and suffers a lot when this number increases. Eclat/LCM and PrefRec seem more resistant to cases where the number of frequent 1-items is large. This will be confirmed with the experiment on synthetic datasets.
Table 3 The Bernoulli variables in the dataset In1000N100Me163

| Number of variables (%) | B(\(p\)) |
|-------------------------|----------|
| 20                      | 0.05 ≤ \(p\) ≤ 0.13 |
| 35                      | 0.13 ≤ \(p\) ≤ 0.16 |
| 35                      | 0.16 ≤ \(p\) ≤ 0.20 |
| 10                      | 0.20 ≤ \(p\) ≤ 0.40 |

Synthetic Datasets

We now present the comparisons on synthetic datasets, that is, on datasets constructed by simulation. More precisely, for each dataset, we simulate an \(N\)-sample \((x_i^{(1)}, \ldots, x_i^{(n)})\) with \(i \leq N\) of the random variable \((X^{(1)}, \ldots, X^{(n)}) \in \{0, 1\}^n\). For all the datasets we have simulated, the number of individuals is \(N = 100000\). Four datasets have a number of items \(n = 1000\). Two of these four datasets are constructed with independent variables \(X^{(j)}\), \(1 \leq j \leq n\), and the other two with variables \(X^{(j)}\), \(1 \leq j \leq n\), defined with the help of a time series.

To confirm the importance of the number of frequent 1-items, and thus also that of the parameter \(n\), we added two datasets with respectively \(n = 4000\) and \(n = 6000\). These added datasets are also obtained using a time series.

In each dataset, the transaction sizes remain quite close. In the names of the datasets, \(n\) is the number of items, \(N\) is the number of individuals in thousands and Me is the averaged number of items per transaction.

The two independent datasets are generated as follows. For each of the \(n\) variables, we first choose randomly a value \(p\) in some given interval \(J \subset [0, 1]\) and then we generate an \(N\)-sample of the Bernoulli distribution \(B(\(p\))\). The chosen intervals \(J\) are given in Tables 2 and 3. For example, line 1 in Table 2 says that for 30\% of the \(n\) variables, the parameter \(p\) of each variable has been chosen uniformly between 0.005 and 0.08.

Table 4 Number of frequent items and frequent itemsets according to the minSup value for synthetic datasets with \(n = 1000\)

| minSup | 0.004 | 0.005 | 0.006 | 0.007 | 0.008 | 0.009 | 0.01 |
|--------|-------|-------|-------|-------|-------|-------|-----|
| items  | 1000  | 1000  | 995   | 989   | 986   | 980   | 977 |
| In1000N100Me103 | 4009246 | 1698133 | 940501 | 567756 | 382110 | 305383 | 263334 |
| minSup | 0.015 | 0.02  | 0.025 | 0.03  | 0.035 | 0.04  | 0.045 |
| items  | 1000  | 1000  | 1000  | 1000  | 1000  | 1000  | 1000 |
| In1000N100Me163 | 4212703 | 1493372 | 662189 | 339070 | 187319 | 120438 | 80752 |
| minSup | 0.01  | 0.015 | 0.02  | 0.025 | 0.03  | 0.035 | 0.04 |
| items  | 1000  | 1000  | 1000  | 1000  | 1000  | 1000  | 1000 |
| ARn1000N100Me103 | 9585055 | 1596769 | 493628 | 187578 | 88097 | 47300 | 29748 |
| minSup | 0.025 | 0.03  | 0.035 | 0.04  | 0.045 | 0.05  | 0.055 |
| items  | 1000  | 1000  | 1000  | 1000  | 1000  | 1000  | 1000 |
| ARn1000N100Me163 | 7860411 | 2878483 | 1303059 | 650431 | 348069 | 203940 | 132006 |
Fig. 9 Running time in seconds for synthetic datasets with large \( n \)

with \( s = 1.7 \). The thresholds \( s \) were chosen such that for \( n = 1000 \), the dependent synthetic datasets have the same \( Me \) as the independent synthetic datasets.

With \( n = 4000 \) and \( s = 1.7 \), we generate ARn-4000N100Me658 and with \( n = 6000 \) and \( s = 1.7 \) we generate ARn6000N100Me987.

Table 4 below is organized like Table 1. It gives for the four datasets with \( n = 1000 \), the number of frequent items and the number of frequent itemsets according to the minSup values. Figure 8 presents the execution times to make these extractions.

In view of these results with \( n = 1000 \), it is clearly seen that Apriori and Fp-Growth are very inefficient compared to Eclat/LCM and PrefRec. We also see that PrefRec is slightly more efficient than Eclat/LCM in these four datasets.

Let us go further and now consider the case where \( n \) is greater.

The results for the synthetic datasets with \( n = 4000 \) and \( n = 6000 \) are given in Table 5 and Fig. 9 below. In Fig. 9, only the times taken by PrefRec and Eclat/LCM are shown. The execution times of Apriori and Fp-Growth are too long to be represented.

In these last two experiments, PrefRec seems, once again, a little more efficient than Eclat/LCM. This confirms the results of the previous four experiments.

For the 10 datasets we used, we can conclude that PrefRec is more efficient than Apriori in all cases and that, in general, it supports the comparison very well with the other algorithms. In the case when the number of frequent items is very low, it has a little more difficulty than Fp-Growth and Eclat/LCM. But as soon as this number is higher it becomes more efficient than Fp-Growth and quite comparable with Eclat/LCM. Note also that PrefRec seems particularly effective in the case of time series. This is probably because it is recursive.

### Applications of Recursion Properties

In this part, we present applications that can be made thanks to the recursion properties of PrefRec. These applications show the important and innovative properties of PrefRec. They cannot be performed with the other algorithms. A major advantage of the recursion properties is that all the variables of the database do not need to be definitively entered before the start of the algorithm: the data can be processed as and when they arrive, which is particularly useful for instance in the case of time series. The applications presented in this section are therefore carried out with non-fixed (or moving) databases, that is, databases whose set of 1-items does change during the process. Of course, there is then no pre-processing of the database.

### Adding Variables

We present here the results of the experiment consisting in adding new variables to a database. An initial database is given. Then, new variables are introduced into the model; we measure the time needed to update the list of frequent itemsets.

We consider the independent case and the dependent case. In both cases, the notation \( re \) indicates the number of variables introduced into the model.

In the independent case, the initial bases considered are In1000N100Me103 and In1000N100Me163. When we introduce a single variable, \( re1 \), it is a Bernoulli variable with parameter \( p \) chosen randomly in a given interval. As an example the title \( re1[0.08:0.12] \) means that a Bernoulli variable with a parameter \( p \), chosen randomly in \([0.08, 0.12]\), was added after the start of the algorithm: the data can be processed as and when they arrive, which is particularly useful for instance in the case of time series. This is probably because it is recursive.

### Applications of Recursion Properties

In this part, we present applications that can be made thanks to the recursion properties of PrefRec. These applications show the important and innovative properties of PrefRec. They cannot be performed with the other algorithms. A major advantage of the recursion properties is that all the variables of the database do not need to be definitively entered before the start of the algorithm: the data can be processed as and when they arrive, which is particularly useful for instance in the case of time series. The applications presented in this section are therefore carried out with non-fixed (or moving) databases, that is, databases whose set of 1-items does change during the process. Of course, there is then no pre-processing of the database.

### Adding Variables

We present here the results of the experiment consisting in adding new variables to a database. An initial database is given. Then, new variables are introduced into the model; we measure the time needed to update the list of frequent itemsets.

We consider the independent case and the dependent case. In both cases, the notation \( re \) indicates the number of variables introduced into the model.

In the independent case, the initial bases considered are In1000N100Me103 and In1000N100Me163. When we introduce a single variable, \( re1 \), it is a Bernoulli variable with parameter \( p \) chosen randomly in a given interval. As an example the title \( re1[0.08:0.12] \) means that a Bernoulli variable with a parameter \( p \), chosen randomly in \([0.08, 0.12]\), was added after the start of the algorithm: the data can be processed as and when they arrive, which is particularly useful for instance in the case of time series. This is probably because it is recursive.

### Applications of Recursion Properties

In this part, we present applications that can be made thanks to the recursion properties of PrefRec. These applications show the important and innovative properties of PrefRec. They cannot be performed with the other algorithms. A major advantage of the recursion properties is that all the variables of the database do not need to be definitively entered before the start of the algorithm: the data can be processed as and when they arrive, which is particularly useful for instance in the case of time series. The applications presented in this section are therefore carried out with non-fixed (or moving) databases, that is, databases whose set of 1-items does change during the process. Of course, there is then no pre-processing of the database.

### Adding Variables

We present here the results of the experiment consisting in adding new variables to a database. An initial database is given. Then, new variables are introduced into the model; we measure the time needed to update the list of frequent itemsets.

We consider the independent case and the dependent case. In both cases, the notation \( re \) indicates the number of variables introduced into the model.

In the independent case, the initial bases considered are In1000N100Me103 and In1000N100Me163. When we introduce a single variable, \( re1 \), it is a Bernoulli variable with parameter \( p \) chosen randomly in a given interval. As an example the title \( re1[0.08:0.12] \) means that a Bernoulli variable with a parameter \( p \), chosen randomly in \([0.08, 0.12]\), was added after the start of the algorithm: the data can be processed as and when they arrive, which is particularly useful for instance in the case of time series. This is probably because it is recursive.

### Applications of Recursion Properties

In this part, we present applications that can be made thanks to the recursion properties of PrefRec. These applications show the important and innovative properties of PrefRec. They cannot be performed with the other algorithms. A major advantage of the recursion properties is that all the variables of the database do not need to be definitively entered before the start of the algorithm: the data can be processed as and when they arrive, which is particularly useful for instance in the case of time series. The applications presented in this section are therefore carried out with non-fixed (or moving) databases, that is, databases whose set of 1-items does change during the process. Of course, there is then no pre-processing of the database.

### Adding Variables

We present here the results of the experiment consisting in adding new variables to a database. An initial database is given. Then, new variables are introduced into the model; we measure the time needed to update the list of frequent itemsets.

We consider the independent case and the dependent case. In both cases, the notation \( re \) indicates the number of variables introduced into the model.

In the independent case, the initial bases considered are In1000N100Me103 and In1000N100Me163. When we introduce a single variable, \( re1 \), it is a Bernoulli variable with parameter \( p \) chosen randomly in a given interval. As an example the title \( re1[0.08:0.12] \) means that a Bernoulli variable with a parameter \( p \), chosen randomly in \([0.08, 0.12]\), was added after the start of the algorithm: the data can be processed as and when they arrive, which is particularly useful for instance in the case of time series. This is probably because it is recursive.
In the dependent case, the initial bases considered are ARn1000N100Me103 and ARn1000N100Me163. The added variables are obtained from the time series. In the case re1, the added variable is $X^{(n+1)}$. In the case re10 (resp. re100) the added variables are $X^{(n+1)}, \ldots, X^{(n+10)}$ (resp. $X^{(n+1)}, \ldots, X^{(n+100)}$).

---

**Table 6** Averaged number of additional frequent itemsets after adding variables to the database in the independent case

| In1000N100Me103 | 0.005 | 0.007 | 0.01 | 0.012 | 0.015 | 0.02 | 0.025 |
|-----------------|-------|-------|------|-------|-------|------|-------|
| re1[0.005:0.08] | 345   | 203   | 70   | 27    | 12    | 1    | 1     |
| re1[0.08:0.12]  | 1547  | 733   | 523  | 350   | 158   | 44   | 12    |
| re1[0.12:0.15]  | 4348  | 1296  | 717  | 622   | 448   | 146  | 63    |
| re1[0.15:0.25]  | 27774 | 7174  | 1576 | 958   | 689   | 510  | 303   |
| re10            | 48929 | 13170 | 5753 | 3858  | 2466  | 1116 | 605   |
| re100           | 57068 | 30411 | 21821| 18933 | 14189 | 6877 | 2787  |
| In1000N100Me163 | 0.020 | 0.022 | 0.025| 0.030 | 0.035 | 0.040| 0.045 |
| re1[0.005:0.08] | 192   | 145   | 97   | 45    | 26    | 14   | 6     |
| re1[0.08:0.12]  | 959   | 574   | 352  | 223   | 122   | 81   | 57    |
| re1[0.12:0.15]  | 2433  | 1675  | 914  | 389   | 267   | 170  | 107   |
| re1[0.15:0.25]  | 21386 | 14733 | 9198 | 4471  | 2303  | 1388 | 870   |
| re10            | 41252 | 27075 | 16355| 6825  | 3981  | 3072 | 1638  |
| re100           | 472422| 322674| 200038|101111|51621|28968|19351 |

---

**Fig. 10** Execution time in seconds to add variables to the database In1000N100Me103

**Fig. 11** Execution time in seconds to add variables to the database In1000N100Me163
Table 7 Averaged number of additional frequent itemsets after adding variables to the database in the AR case

| ARn1000N100Me103 | 0.015 | 0.0175 | 0.020 | 0.025 | 0.030 | 0.035 | 0.040 |
|------------------|-------|--------|-------|-------|-------|-------|-------|
| re1              | 1682  | 882    | 521   | 197   | 92    | 50    | 35    |
| re10             | 16925 | 8931   | 5310  | 1999  | 920   | 502   | 318   |
| re100            | 172163| 91032  | 54143 | 20036 | 9567  | 5195  | 3200  |
| ARn1000N100Me163 | 0.035 | 0.0375 | 0.040 | 0.045 | 0.050 | 0.055 | 0.060 |
| re1              | 1183  | 813    | 607   | 329   | 188   | 126   | 76    |
| re10             | 12627 | 8639   | 6336  | 3388  | 1974  | 1290  | 775   |
| re100            | 141364| 91236  | 65233 | 34701 | 20063 | 13400 | 8651  |

![Graphs](image-url)

Fig. 12 Execution time in seconds to add variables to the database ARn1000N100Me103 or ARn1000N100Me163

Because they are not intended for this, Eclat/LCM, Apriori and Fp-Growth must be restarted for each added variable, unlike PrefRec which was defined and built around recursive logic. Therefore we only present the calculation times of PrefRec, the calculation times of the others algorithms being too long to be represented.

Since the results are random, we conducted 50 simulations. The results presented are averages over the 50 simulations. Tables 6 (for the independent datasets) and 7 (for the dependent datasets) indicate the number of new frequent itemsets after adding variables. Figs. 10 and 11 (for the independent datasets) and Fig. 12 (for the dependent datasets) give the running time as a function of minSup.

Table 6 shows that the number of added frequent itemsets increases as the frequency of the added variable increases. In the AR case (Table 7), the variables having the same distribution law, the addition of 10 variables produces a number of frequent itemsets 10 times greater than the addition of a single variable.

Likewise, Figs. 10, 11, and 12 show that the execution time depends on the frequency of the added variable, and on the minSup value. It also seems that if the number of added variables is multiplied by 10, then the execution time is also multiplied by 10. Above all, these figures show that PrefRec is particularly effective for updating the algorithm when a new variable (or more) is taken into account.

The Moving FIM Application

We now present four examples of the Moving FIM application. The first step of the Moving FIM application is the construction of the tree of the frequent itemsets based on the first $q$ frequent items. The second step consists in actualizing the tree: when a new frequent item arrives, the algorithm adds the frequent itemsets that contain this new frequent item, and delete all the frequent itemsets that contain the most right child of ($\emptyset$). The number of frequent items, therefore, remains constant and equal to $q$. In the experiment, we consider a fixed database. Then, to make the moving FIM work, we process the variables of this database one after the other.

The databases we used are the four previous independent and AR with $n = 1000$.

We considered two values of $q$: 100 and 300. Moreover, in each experiment, we performed $Q = 100$ frequent item replacements.

The results given in Tables 8 and 9 concern only the second step of the Moving FIM application, that is, the actualization of the tree. Both tables give the number of frequent itemsets added (freq-add), the number of frequent itemsets deleted (freq-del), and the time necessary for the application. All the values given in these tables are the averages obtained over 50 simulations.
Tables 8 and 9 show that the variations associated with adding and removing a variable depend mainly on the values of \( \text{minSup} \) and \( q \). Moreover, the averaged number of frequent itemsets added and deleted is much higher in the case of dependent database than in the case of independent database. But, for both types of databases, there is a stability in the number of frequent itemsets during this application, the number of added frequent itemsets being always close to that of deleted frequent itemsets.

PrefRec is very performant for such an actualization of the tree of the frequent itemsets. This is due to the nature of the underlying tree.
Conclusion

In this paper, we presented PrefRec, a new algorithm for fast mining of frequent itemsets and discovering all significant association rules in large datasets. The logic of the algorithm is based on a function which, starting from a tree of frequent itemsets, allows to find the new tree of frequent itemsets when a new item is added to the base. The main property of the algorithm is its recursiveness with respect to the sequence of variables. This makes it very useful when the processed dataset is not fixed and can evolve by adding new variables. Another important property of the algorithm is that it gives the list of frequent itemsets in a very useful tree structure. In particular, this makes it possible to remove from the list of frequent itemsets, in a very simple and very fast way, the first item of the base and all the frequent itemsets which contain it. This makes PrefRec a very efficient algorithm in the case of a moving base, i.e. a base where the sequence of variables changes by removing the first and adding a new one at the end of the sequence. A large set of experiments have been carried out. First, we compared the performance of PrefRec and that of Eclat/LCM, Fp-Growth and Apriori on fixed databases. The experiments made in “Comparison with Non-recursive Algorithms” show that PrefRec supports the comparison well. Then, we evaluated the recursion properties of PrefRec. To this end, we considered non-fixed databases, that is, databases whose set of 1-items changes during the process. Eclat/LCM, Fp-Growth and Apriori are not designed to deal with this type of database, whereas the experiments made in “Applications of Recursion Properties” show that PrefRec is very efficient.

Acknowledgements We would like to thank the Maisons du Monde data team for the numerous discussions we had on the various applications of the PrefRec algorithm.

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Agrawal R, Shafer J. Parallel mining of association rules: design, implementation, and experience. IEEE Trans Knowl Data Eng. 1996;8:962–9.
2. Agrawal R, Srikant R. Fast algorithms for mining association rules. In: Proceedings of the 1994 international conference on very large data bases (VLDB 94) (Santiago, Chile); 1994. pp. 487–499.
3. Agrawal R, Imielinski T, Swami A. Mining association rules between sets of items in large databases. In: Proceedings of the 1993 ACM-SIGMOD international conference on management of data (SIGMOD’93) (Washington, DC); 1993. pp. 207–216.
4. Agrawal R, Mannila H, Srikant R, Toivonen H, Verkamo A. Fast discovery of association rules. In: Advances in knowledge discovery and data mining, AAAI/MIT Press; 1996. pp. 307–328.
5. Borgelt C. Frequent item set mining WIREs. Data Mining Knowl Discov. 2012;2:437–56.
6. Brin S, Motwani R, Ullman JD, Tsur S. Dynamic itemset counting and implication rules for market basket analysis. In: Proceeding of the 1997 ACM-SIGMOD international conference on management of data (SIGMOD’97) (Tucson, AZ); 1997. pp. 255–264. https://doi.org/10.1145/253260.253325.
7. Cheung DW, Han J, Ng V, Fu A, Fu Y. A fast distributed algorithm for mining association rules. In: Proceeding of the 1996 international conference on parallel and distributed information systems (Miami Beach, FL); 1996. pp. 31–42.
8. Cheung DW, Han J, Ng V, Wong CY. Maintenance of discovered association rules in large databases: an incremental updating technique. In: Proceeding of the 1996 international conference on data engineering (ICDE’96) (New Orleans, LA); 1996. pp. 106–114.
9. Han J, Pei J, Yin Y. Mining frequent patterns without candidate generation. In: Proceeding of the 2000 ACM-SIGMOD international conference on management of data (SIGMOD’00) (Dallas, TX); 2000. pp. 1–12.
10. Han J, Cheng H, Xin D, Yan X. Frequent pattern mining: current status and future directions. Data Min Knowl Discov. 2007;15:55–86.
11. Huang L, Chen H, Wang X, Chen G. A fast algorithm for mining association rules. Comput Sci Technol. 2000;15:619–24.
12. Luna JM, Fournier-Viger P, Ventura S. Frequent itemset mining. Frequent itemset mining: a 25 Years Review; 2019. https://doi.org/10.1002/widm.1329.
13. Park JS, Chen MS, Yu PS. An effective hash-based algorithm for mining association rules. In: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (San Jose, California, USA); 1995. pp. 175–186. https://doi.org/10.1145/568271.223813.
14. Sarawagi S, Thomas S, Agrawal R. Integrating association rule mining with relational database systems: alternatives and implications. In: Proceeding of the 1998 ACM-SIGMOD international conference on management of data (SIGMOD’98) (Seattle, WA); 1998. pp. 343–354.
15. Sivaswamy A, Omiecinski E, Navathe S. An efficient algorithm for mining association rules in large databases. In: Proceeding of the 1995 international conference on very large data bases (VLDB’95) (Zurich, Switzerland); 1995. pp. 432–444.
16. Toivonen H. Sampling large databases for association rules. In: Proceeding of the 1996 international conference on very large data bases (VLDB’96) (Bombay, India); 1996. pp. 134–145.
17. Uno T, Asai T, Uchida Y, Arimura H. LCM: an efficient algorithm for enumerating frequent closed item sets. In: Proceedings of the workshop on frequent item set mining implementations (FIMI 2003) (Melbourne, FL). TU Aachen, Germany: CEUR Workshop Proceedings 90; 2003.
18. Zaki MJ. Scalable algorithms for association mining. IEEE Trans Knowl Data Eng. 2000;12:372–90.
19. Zaki MJ, Parthasarathy S, Ogihara M, Li W. Parallel algorithm for discovery of association rules. Data Mining Knowl Discov. 1997;1:343–73.

Publisher’s Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.