Mining Maximal Frequent Itemsets by a Boolean Based Approach

Ansaf Salleb[1], **Zahir Maazouzi**[1], **Christel Vrain** [1]

Abstract.
We propose a new boolean based approach for mining frequent patterns in large transactional data sets. A data set is viewed as a truth table with an output boolean function. The value of this function is set to one if the corresponding transaction exists in the data set, zero otherwise. The output function represents a condensed form of all the transactions in the data set and is represented by an efficient compact data structure. It is then explored with a depth first strategy to mine maximal frequent itemsets. We have developed a prototype, and first experiments have shown that it is possible to rely on a condensed representation based on boolean functions to mine frequent itemsets from large databases.

Keywords: Maximal Frequent Itemsets, Association Rules, Data Mining, Boolean Algebra, Binary Decision Diagrams.

1 Introduction

Mining frequent itemsets is an active area in data mining that aims at searching interesting relationships between items in databases. It can be used to address to a wide variety of problems such as discovering association rules, sequential patterns, correlations and much more. A transactional database is a data set of transactions, each composed of a set of items, called an itemset. An itemset is frequent when it occurs ”sufficiently” in the database, that is to say in at least a certain proportion of all the transactions, with respect to a given threshold. Since finding frequent itemsets is a time consuming task, especially when databases are dense or when long patterns of itemsets emerge, this has been extended to mining maximal frequent itemsets, i.e. mining the longest frequent itemsets.

We propose a new approach based on properties of boolean algebra since a transactional database can be seen as a truth table with an output function. This function is loaded in main memory by using a Binary Decision Diagram (BDD) introduced in [5, 4]. A tree is then explored on this function recursively in order to mine maximal frequent itemsets. Moreover, the search space is reduced by a preprocessing step that keeps only the frequent variables according to the support threshold, and that reorders them in an increasing order. The present work illustrates that efficient data structures and algorithms manipulating boolean functions coming from digital logic design can be interesting for such data mining task. Our first prototype and experiments have shown that it is a promising research issue.

This paper is organized as follows: in section 2, we describe the basic concepts for mining frequent itemsets. Some theoretical aspects of vectorial boolean algebra, on which our approach relies, are given in section 3. In section 4, we describe our algorithm to compute maximal frequent itemsets. Details on implementation and experimental tests are discussed in section 5. In section 6, we briefly give related works. Finally, we conclude with a summary of our approach, perspectives and extensions of this work. Let us notice that for lack of space, all proofs are omitted.

2 Preliminaries

Let $\mathcal{I} = \{x_1, x_2, ..., x_n\}$ be a set of items. Let $\mathcal{T}$ be a data set of transactions, where each transaction is a set of items belonging to $\mathcal{I}$, called an itemset. The total number of itemsets grows exponentially with $|\mathcal{I}|$. In order to restrict this number, a minimum support threshold *MinSupp* is specified, and only frequent itemsets, with respect to this threshold are considered. Let $\mathcal{X}$ be an itemset. The support of $\mathcal{X}$ is $Supp(\mathcal{X}) = |\{t_i \in \mathcal{T} / \mathcal{X} \subseteq t_i\}|$. $\mathcal{X}$ is frequent if its support is higher or equal to *MinSupp*. This means that $\mathcal{X}$ is frequent when it is a subset of at least *MinSupp* transactions in $\mathcal{T}$.
So, mining frequent itemsets is finding all itemsets that satisfy the minimum support threshold. A frequent itemset is maximal when all its subsets are frequent but also when all its supersets are infrequent.

Example 1 *Market Basket Analysis Example*

Tid	ItemSet
1	x_2
2	x_3
3	x_4
4	x_1, x_4
5	x_2, x_4
6	x_3, x_4
7	x_1, x_2, x_4
8	x_2, x_3, x_4
9	x_1, x_2, x_3, x_4

Table 1. Transactional table

x_1	x_2	x_3	x_4	f
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

Table 2. Truth table

The search space forms a lattice where nodes represent all possible subsets of $\mathcal{I}$ as illustrated in Figure 1 . Looking at the support of each

[1] LIFO, Université d'Orléans, B.P. 6759, F-45067 Orléans, France. Email:{salleb, maazouzi, cv}@lifo.univ-orleans.fr

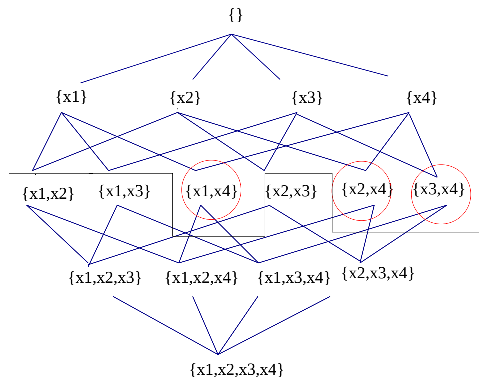

Figure 1. Itemsets lattice of Table 1

node is infeasible. So exploring efficiently the search space means restricting the number of nodes to visit in order to find all the frequent itemsets as soon as possible. Lattice nodes can be traversed in either a depth first search (DFS) or a breadth first search (BFS).

In the example above, we consider a set of items $\mathcal{I} = \{x_1, x_2, x_3, x_4\}$ and the transactional database given in Table 1 composed of nine transactions each one has a unique identifier *Tid*. The search space is visualized as a lattice, where each node is an itemset. For a MinSupp $= 3$ transactions, the frequent itemsets are those above the border. Maximal frequent itemsets are circled nodes.

3 Vectorial aspects of boolean algebra

Assuming readers are familiar with basic notions on boolean algebra [11, 20, 19, 13], we begin in this section by recalling a few definitions on boolean vectors, and some useful operators on these objects [15].

3.1 Boolean vectors

A boolean vector $\overrightarrow{v} = (v_1, v_2, \ldots, v_n)$ is any n-tuple of values belonging to the set $\{0,\ 1\}$. Let $\overrightarrow{\mathbb{E}}^n$ be the set of all boolean vectors of size 2^n. For example, $\overrightarrow{\mathbb{E}}^2 = \{$(0000), (0001), (0010), (0011), (0100), (0101), (0110), (0111), (1000), (1001), (1010), (1011), (1100), (1101), (1110), (1111)$\}$. The dimension of a vector of $\overrightarrow{\mathbb{E}}^n$ is n, and its size is 2^n. A truth table $\vec{\mathbb{T}}^n$ of dimension n is made up of n boolean vectors, each of them belonging to $\vec{\mathbb{E}}^n$. We write $\vec{\mathbb{T}}^n = [\overrightarrow{e_1^n}, \ldots, \overrightarrow{e_n^n}]$. For each index j, $1 \leq j \leq n$, $\overrightarrow{e_j^n}$ is the j-th vector of $\vec{\mathbb{T}}^n$. A truth table $\vec{\mathbb{T}}^n = [\overrightarrow{e_1^n}, \ldots, \overrightarrow{e_n^n}]$ is recursively built as follows: $[\vec{0}^{n-1} \odot \vec{1}^{n-1},\ \overrightarrow{e_1^{n-1}} \odot \overrightarrow{e_1^{n-1}},\ \overrightarrow{e_2^{n-1}} \odot \overrightarrow{e_2^{n-1}}, \ldots, \overrightarrow{e_{n-1}^{n-1}} \odot \overrightarrow{e_{n-1}^{n-1}}]$ where $\vec{0}^n$ and $\vec{1}^n$ correspond respectively to the null and identity vectors of dimension n, and $\odot$ represents the concatenation of vectors. For instance, $(0011) \odot (0101) = (00110101)$.

In the following, we use the vectorial product between two boolean vectors: if $\vec{v} = (v_1, \ldots, v_k)$ and $\vec{w} = (w_1, \ldots, w_k)$, then $\vec{v} \cdot \vec{w} = (v_1 \cdot w_1, \ldots, v_k \cdot w_k)$. It consists in performing an *and* operation.

A vectorial function with n variables is a pair whose first element is a truth table and the second one a goal vector of dimension n representing the output [15]. Table 2 shows the truth table of a function with 4 variables whose goal is $f = (0111110101000101)$.

3.2 Boolean vectors and transactional databases

Let us now explain the correspondence between a transactional database and a vectorial function. In our approach, we represent an item with a boolean variable. Given n items, $x_1, \ldots, x_n$, a set of transactions $\mathcal{T}$ is represented by a vectorial function with n variables.

For instance, the basket analysis example (Example 1) is represented by the function whose truth table is shown in Table 2.

Each line of the truth table corresponds to a possible transaction. The function f is true for all the transactions belonging to $\mathcal{T}$, and false for the other ones. Since the structure of a truth table is fixed, giving a boolean vector corresponding to f is sufficient to express the set of transactions $\mathcal{T}$. Let us notice that we deal with transactions that exist only once in $\mathcal{T}$. The approach presented in this paper follows this restriction, but our work can be extended to deal with general sets of transactions.

We also represent an itemset by a string of length n built over the alphabet $\{1, \varepsilon\}$. For instance, the string $11\varepsilon1$ corresponds to the itemset $\{x_1, x_2, x_4\}$, since the variable x_3 is not present in the itemset. We call this new representation the *hat representation*. The set of hat representations of n variables is denoted by $\hat{\mathbb{H}}^n$.

Assuming that the variables are ordered by an arbitrary order, it is easy to see that there is a bijection between these two representations. Indeed, let $\mathcal{X}$ be an itemset, the transformation consists just in associating the value 1 to the item (or variable) x_i if it belongs to $\mathcal{X}$, and ε if the variable is not present. By this bijection, we can extend the inclusion ordering on sets by the following ordering on hat representation of $\hat{\mathbb{H}}^n$.

Definition 1 (Extension of inclusion ordering on $\hat{\mathbb{H}}^n$) *Let $\hat{p}$ and $\hat{q}$ be elements of $\hat{\mathbb{H}}^n$ such that $\hat{p} = c_1c_2\ldots c_n$ and $\hat{q} = d_1d_2\ldots d_n$. The order $\preceq$ is recursively defined as follows:*
1. $1 \preceq \varepsilon, 1 \preceq 1, \varepsilon \preceq \varepsilon$;
2. $\hat{p} \preceq \hat{q}$ *iff* $\forall i \quad c_i \preceq d_i$.
This order $\preceq$ can obviously be restricted to $\prec$ as follows :
1. $1 \prec \varepsilon$;
2. $\hat{p} \prec \hat{q}$ *iff* $\forall i \quad c_i \preceq d_i$ *and* $\exists\, j \quad c_j \prec d_j$.

Example 2

For $n = 4$, we have $111\varepsilon \prec \varepsilon\varepsilon1\varepsilon$. The corresponding itemsets are respectively $\{x_1, x_2, x_3\}$ and $\{x_3\}$, and we have $\{x_1, x_2, x_3\} \supset \{x_3\}$.

With this correspondence, we have: if $\mathcal{X}_1$ and $\mathcal{X}_2$ are two itemsets, and $\hat{p}_1$, $\hat{p}_2$ are their corresponding hat representations (products), then $\mathcal{X}_1 \supset \mathcal{X}_2$ iff $\hat{p}_1 \prec \hat{p}_2$.

Let us notice that if an itemset represented by a hat product $\hat{p}$ is not frequent then all itemsets represented by a hat product $\hat{q}$ such that $\hat{q} \preceq \hat{p}$ are also not frequent, since the frequency of the itemset associated to $\hat{q}$ is less than the frequency associated to $\hat{p}$. Consequently, in the following, terms given below will be considered as equivalent:

- an item and a boolean variable ;
- an itemset and a hat representation ;
- an itemset $\mathcal{X}_1$ contains another itemset $\mathcal{X}_2$ ($\mathcal{X}_2 \subseteq \mathcal{X}_1$) is equivalent to say that the hat representation $\hat{p}_1$ (corresponding to $\mathcal{X}_1$) is smaller than $\hat{p}_2$ (corresponding to $\mathcal{X}_2$) with respect to the $\preceq$ ordering ;
- a set of transactions is equivalent to a boolean vector.

4 A new approach to compute maximal frequent itemsets

As stated in section 3.2, we represent a set of transactions with a boolean vector f. The itemsets are explored using a binary tree whose nodes are boolean vectors, and the edges are respectively labelled with 1 or ε. Let us consider again the basket analysis example given in Example 1. The corresponding truth table is shown in Table 2, and the associated tree is illustrated in Figure 2. For sake of space, only a part of the tree is drawn.

The following function $\mathcal{MFI}temsets$ returns a set of hat representations corresponding to the maximal frequent itemsets associated to the transactions given in the first parameter. In this algorithm we introduce two new notations about set handling. The notation $2^{\hat{\mathbb{H}}}$ is the set of all the subsets of $\hat{\mathbb{H}}$. The set $\{emptyhat\}$ is a set that contains an empty hat string. It is different of an empty set $\emptyset$ (or $\{\}$). We denote by n the total number of items in the dataset. The first call of $\mathcal{MFI}temsets$ is performed with a level l equal to 0.

```
function $\mathcal{MFI}temsets(s : \overrightarrow{\mathbb{E}}^n;\ l, minsup : int;\ \mathcal{P} : 2^{\hat{\mathbb{H}}}) : 2^{\hat{\mathbb{H}}}$
begin
    if $frequency(s) < minsup$ then return $\emptyset$
    elif $ContainsOne(\mathcal{P})$ return $\emptyset$
    elif $l = n$ and $\mathcal{P} = \{emptyhat\}$ then return $\emptyset$
    elif $l = n$ and $\mathcal{P} = \emptyset$ then return $\{emptyhat\}$
    else
        $s_1 := \overrightarrow{e}_{l+1} \cdot s$
        $\mathcal{P}_1 := \{\hat{p} \ / \ 1\hat{p} \in \mathcal{P}\}$
        $\mathcal{Q}_1 := \mathcal{MFI}temsets(s_1, l+1, minsup, \mathcal{P}_1)$
        $s_\varepsilon := s$
        $\mathcal{P}_\varepsilon := \{\hat{p} \ / \ c\hat{p} \in \mathcal{P}$ and $c \in \{1, \varepsilon\}\} \cup \mathcal{Q}_1$
        $\mathcal{Q}_\varepsilon := \mathcal{MFI}temsets(s_\varepsilon, l+1, minsup, \mathcal{P}_\varepsilon)$
        return $\{1\hat{q} \ / \ \hat{q} \in \mathcal{Q}_1\} \cup \{\varepsilon\hat{q} \ / \ \hat{q} \in \mathcal{Q}_\varepsilon\}$
    end if
end
```

This algorithm recursively explores a part of the whole tree shown in Figure 2. In each path (represented by a hat representation), an edge labelled with 1 corresponds to the fact that the current variable is fixed. When the label is equal to ε, the variable is not chosen. The level l given in the second parameter identifies the current variable dealt with. The boolean function stored at a node s gives the transactions that contains the variables set to 1 from the root to that node. In order to know the maximal itemsets already discovered, we associate to each node a set $\mathcal{P}$ which contains these maximal itemsets.

The algorithm can be divided into two parts: the stop tests and the recursive calls. Let us begin with the core of the program. There are always two recursive calls respectively on son s_1 linked to its father with an edge labelled by 1, and on son s_ε linked with an edge ε. The latest son s_ε means that the current variable $l+1$ (since the first level is 0) is not fixed. Thus, all transactions are kept, that is why we have always $s_\varepsilon = s$. On the other hand, s_1 is computed by performing a vectorial product between the father s and the vector $\vec{e}_{l+1}$ corresponding to the current variable. Indeed, this vector is the $(l+1)th$ column in the truth table (see section 3.1). The intuitive idea of this product is that we keep from positions set to 1 in s those that are also set to 1 in $\vec{e}_{l+1}$. After, $\mathcal{P}_1$ and $\mathcal{P}_\varepsilon$ associated respectively to s_1 and s_ε contains the already processed maximal itemsets that can potentially be smaller (w.r.t. $\preceq$) than the path under construction. Let us remark that $\mathcal{P}$ does not contain products of size n (number of items), but products of size $n-l$. Indeed, the prefix is omitted because by construction, it is less (w.r.t. $\preceq$) than the current path. We detail this property as follows. If $c_1 \cdots c_l$ is the path from the root to s, and $\hat{p} \in \mathcal{P}$, then there exists an already computed frequent itemset $\hat{q}$ with $\hat{q} = d_1 \cdots d_l \hat{p}$ and $d_1 \cdots d_l \preceq c_1 \cdots c_l$. The inheritance is performed in two ways. The son s_1 inherits only from its father s, but the son s_ε inherits from its father s and also from the result of the recursive calls of its brother s_1. The set of itemsets $\mathcal{P}_1$ associated to the son s_1 contains the tail of each element of the set $\mathcal{P}$ associated to s for which the head is equal to 1. For instance, if $\mathcal{P} = \{1\varepsilon1\varepsilon, \varepsilon1\varepsilon\varepsilon, 11\varepsilon1\}$ then $\mathcal{P}_1 = \{\varepsilon1\varepsilon, 1\varepsilon1\}$. The inheritance associated to the son s_ε is the union of two sets. The first, built from its father s, contains the tail of each element of $\mathcal{P}$. As above, if $\mathcal{P} = \{1\varepsilon1\varepsilon, \varepsilon1\varepsilon\varepsilon, 11\varepsilon1\}$ then we keep $\{\varepsilon1\varepsilon, 1\varepsilon\varepsilon, 1\varepsilon1\}$. The second set $\mathcal{Q}_1$ is the already computed partial frequent itemsets associated to its brother s_1. This recursive processing is performed until getting a success of a stop test. Once the recursive calls are finished, $\mathcal{Q}_1$ and $\mathcal{Q}_\varepsilon$ corresponding respectively to the maximal itemsets of s_1 and s_ε are returned, and a composition, the reverse operation of the inheritance, is performed by adding 1 to the head of all elements of $\mathcal{Q}_1$ and ε to the head of all elements of $\mathcal{Q}_\varepsilon$. The union of these two new itemsets is returned.

The second part of the algorithm concerns the stop tests. There are four kinds of exit cases. The first one is the classical frequency cut. The function $frequency$ computes the frequency of a boolean vector, representing thus a set of transactions. It is exactly the number of 1 present in the vector. Unlike existing methods, this information is computed on the fly during the generation of vectors. As will be explained in the experimental section, each vector of the tree is a node in the BDD representing the tree. If a vector is not frequent with respect to the minimum support, all the itemsets that can be generated from this node are cut, since they cannot be frequent. Indeed, the function visits all the elements of the lattice presented in Figure 1 following this order : 1111, 111ε, $11\varepsilon1$, $11\varepsilon\varepsilon$, $\ldots$, $\varepsilon\varepsilon\varepsilon1$, $\varepsilon\varepsilon\varepsilon\varepsilon$. This corresponds to all leaves from left to right of the tree in Figure 2. If a node located at path $\varepsilon1$ is not frequent, then $\varepsilon1\varepsilon\varepsilon$ is also not frequent, since this node s and its son s_ε have the same value. Consequently, all the itemsets $\varepsilon1c_3c_4$ such that $c_3, c_4 \in \{1, \varepsilon\}$ are also not frequent, because $\varepsilon111, \varepsilon11\varepsilon, \varepsilon1\varepsilon1, \varepsilon1\varepsilon\varepsilon \preceq \varepsilon1\varepsilon\varepsilon$. Thus the frequency of those four itemsets are less than the frequency of $\varepsilon1\varepsilon\varepsilon$ and by the way of the itemset $\varepsilon1$. At this node, the function stops and does not perform the recursive calls.

The second test is a new original cut based on the notion of inheritance already explained. The function $ContainsOne(\mathcal{P})$ is the key point of this cut. It takes as parameter a set of hat representations and returns true if the set $\mathcal{P}$ contains a hat representation composed only of character 1. In a formal way:

$$ContainsOne(\mathcal{P}) = \begin{cases} true \text{ if } \exists\, \hat{p} = c_1 \cdots c_k \in \mathcal{P} \text{ s.t. } \forall i\ c_i \neq \varepsilon \\ false \text{ otherwise} \end{cases}$$

If $ContainsOne(\mathcal{P})$ is true then there exists $\hat{p} \in \mathcal{P}$ such that $\hat{p} = 1 \cdots 1$ and it will be smaller (with respect to $\preceq$) than any generated $\hat{q}$ from the current node. Indeed, $\hat{p} \preceq \hat{q} = d_1 \cdots d_k$ since $1 \preceq d_i$ for all $d_i \in \{1, \varepsilon\}$. Therefore, in this case we return an empty set corresponding to the fact that there is no maximal frequent itemsets that can be generated from this node.

Finally, the two last tests correspond to the case where a leaf is reached. If a leaf inherits a non empty set $\mathcal{P}$, then the path $\hat{p}$ from the root to this leaf is not maximal. On the other hand, if $\mathcal{P} = \emptyset$, then there is no other already computed maximal itemset $\hat{q}$ less or equal to $\hat{p}$ with respect to $\preceq$. Remark that these two last cases, dealing with leaves, are particular cases of the cut due to the $ContainsOne$ function.

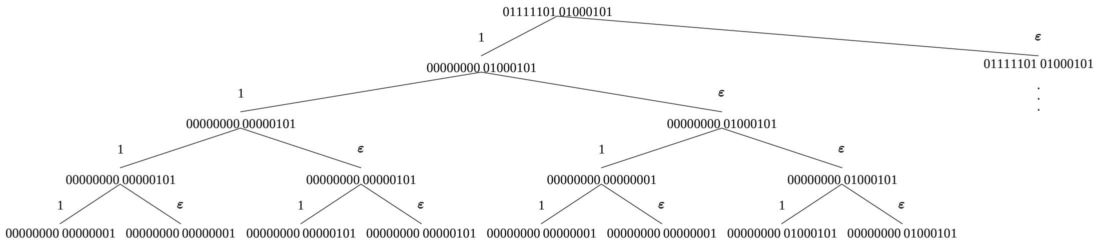

Figure 2. Basket example itemsets represented with binary tree

Example 3

Figure 3 shows the tree generated by $\mathcal{MFI}temsets(0111110101000101, 0, 3, \emptyset)$. The corresponding set of maximal frequent itemsets is $\{1\varepsilon\varepsilon1, \varepsilon1\varepsilon1, \varepsilon\varepsilon11\}$. For sake of space, only the subtree 1 of the root is drawn.

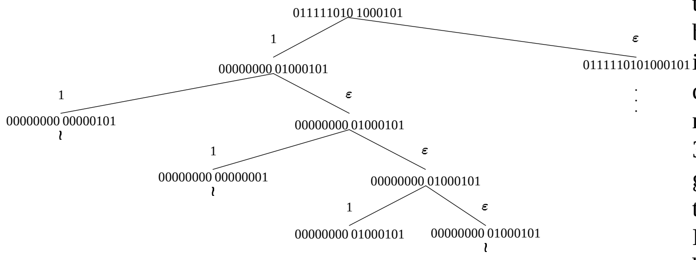

Figure 3. Maximal frequent itemsets of Example 1 with minsup=3

The minimum support is equal to 3, and the value of $\mathcal{P}$, last parameter of $\mathcal{MFI}temsets$ is the empty set since there is no maximal itemsets at the beginning. The symbol ≀ is a cut due to the frequency test or to the success of the test $ContainsOne(\mathcal{P}_\varepsilon)$.

Theorem 1 *Let f be a boolean vector corresponding to a set of transactions $\mathcal{T}$, and M a given minimum support. Calling $\mathcal{MFI}temsets(f, 0, M, \emptyset)$ returns all maximal frequent itemsets and only the maximal frequent itemsets of $\mathcal{T}$.*

5 Implementation and experimental results

We have implemented a prototype in C, where the main data structure used is BDDs (compact Direct Acyclic Graph data structure). Readers interested in BDDs should refer to [5, 4] for more details. Each vector is not stored statically, but is represented by a BDD. All new computed vectors are pointers to their corresponding BDDs, and these vectors share mutually sub-BDDs. The set representing the maximal frequent itemsets is also represented by a BDD. Consequently, the tree built by $\mathcal{MFI}temsets$ function does not really exist and it is just a representation of its execution. Huge data sets used in our experiments have been completely stored in main memory.

The size of the graph, i.e. the number of nodes does not depend on the number of items [7, 8]. We should also notice that the complexity of our algorithm is proportional to the size of the BDD being operated on, and so it is more efficient for compact graphs. A technique to reduce efficiently the size of the graph is variable ordering. Many algorithms have been studied [16, 9, 17] in the field of binary decision diagrams. In our experiments, we have made an ordering $<$ based on their support. A variable $x_i < x_j$ if the support of x_i is less than the support of x_j. This strategy allows us to cut as soon as possible useless items. Another feature of our method is that no scanning is performed to compute the frequency of a vector, since the whole database is stored in main memory. Tables 3 and 4 show experiments made on a Pentium 500 Mhz with 256 Mb of main memory. Table 3 gives time execution for synthetic databases[2] generated by an algorithm designed by the IBM Quest project and described in [3]. In this kind of databases, T represents the average size of a transaction, I the average size of maximal frequent itemsets, and D is the number of transactions. The number of items of each synthetic database exceeds one thousand (1000) and the number of transactions is 100 thousands and 200 thousands transactions. Table 4 corresponds to the Mushroom[3] real database.

6 Related works

During the last years, a wide variety of algorithms for mining frequent itemsets have been proposed. *Apriori* [2, 3] is certainly the most popular algorithm, it uses an iterative way to explore the frequent itemsets. *MaxEclat and MaxClique* [21] are algorithms based on graph theory, they use decompositions of the lattice into sublattices in order to mine maximal frequent itemsets. *MaxMiner* [12] explores maximal frequent itemsets by using a lookahead pruning strategy. *PincerSearch* [14] starts a bottom-up and a top-down search in the lattice at the same time while looking for maximal frequent patterns. *GenMax* [10] uses a backtracking search for enumerating all maximal frequent itemsets and eliminates progressively non-maximal ones. *Mafia* [6] uses a depth first method and some pruning strategies to mine maximal frequent itemsets. *Depth-project* [1] uses a dynamic reordering in order to reduce the search space. A good evaluation of the existing approach in mining maximal frequent itemsets on both dense and sparse databases is given in [10]. Our approach differs from approaches described above, since it is based on a compact representation of the data set. For the time being, our approach is not comparable to the methods cited above, since we do not handle yet datasets with multiple instance transactions.

[2] http://www.almaden.ibm.com/cs/quest/syndata.html
[3] ftp://ftp.ics.edu/pub/machine-learning-databases/mushroom/

7 Conclusion

We present a new approach based on boolean algebra to mine maximal frequent itemsets in transactional databases. We first consider the database as a truth table with an output boolean function. This latter is represented in memory with a graph-based data structure dedicated for boolean functions: Binary Decision Diagrams. This allows an efficient loading of transactions in main memory and so avoid making expensive scans of the database: the output function is sufficient to mine frequent patterns and it is loaded in a preprocessing step in one scan of the database. Moreover, seraching the set of transactions including a given itemset is processed by making iteratively the product of the boolean function by vectors representing the items.

We have already obtained preliminary and encouraging results, but our approach is new and the algorithm can still be improved by integrating cut strategies used in known algorithms such as *GenMax* [10]. However, in our approach we have made the assumption that the data set does not contain several occurrences of a same transaction. In future works, we plan to extend the approach to handle data sets containing many occurrences of transactions. This approach will be tested on a real application for mining association rules in Geographic Information systems [18]. Finally, we notice that our approach allows an interesting extension in mining frequent itemsets with negation.

ACKNOWLEDGEMENTS

We would like to thank ’R´egion Centre’ of France and the BRGM ’Bureau des Recherches G´eologiques et Minières’ for supporting this work. We also wish to thank the anonymous referees for their comments and suggestions for improving this paper.

REFERENCES

[1] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad, ‘Depth fi rst generation of long patterns’, in *6th Int’l Conference on Knowledge Discovery and Data Mining*, pp. 108–118, (August 2000).
[2] R. Agrawal, T. Imielinski, and A. N. Swami, ‘Mining association rules between sets of items in large databases’, *In Proc. of the ACM SIGMOD International Conference on Management of Data*, 207–213, (1993).
[3] R. Agrawal and R. Srikant, ‘Fast algorithms for mining association rules’, *In Proc. 20th Int. Conf. Very Large Data Bases (VLDB)*, 487–499, (1994).
[4] S.B. Akers, ‘Binary decision diagrams’, *IEEE Transactions on Computers*, **27**, 509–516, (1978).
[5] R.E. Bryant, ‘Graph-based Algorithms for Boolean Functions Manipulation’, *IEEE Trans. on Computers*, **C-35**(8), 677–691, (1986).
[6] D. Burdick, M. Calimlim, and J. E. Gehrke, ‘Mafi a: A maximal frequent itemset algorithm for transactional databases’, *In Proceedings of the 17th International Conference on Data Engineering*, (April 2001).
[7] O. Coudert, ‘Two-level logic minimization : An overview.’, *the VLSI Journal*, **17-2**, 97–140, (1994).
[8] O. Coudert and J. C. Madre, ‘A new method to compute prime and essential prime implicants of boolean functions’, in *Proc. Brown/MIT Conference on Advanced Research in VLSI and Parallel Systems*, Cambridge, MA, USA, (March 1992).
[9] M. Fujita, H. Fujisawa, and N. Kawato, ‘Evaluation and improvement of boolean comparison method based on binary decision diagrams’, in *Proc. of IEEE/ACM International Conference on Computer-Aided Design*, pp. 2–5, (1988).
[10] K. Gouda and M. J. Zaki, ‘Effi ciently mining maximal frequent itemsets’, *1st IEEE International Conference on Data Mining*, (November 2001).
[11] P.R. Halmos, *Lectures Notes on Boolean Algebras*, Springer, Berlin, 1974.
[12] R. J. Bayardo Jr., ‘Effi ciently mining long patterns from databases’, *In ACM-SIGMOD Int’l Conf. on Management of Data*, 85–93, (1998).
[13] R. H. Katz, *Contemporary logic design*, Benjamin Cummings/Addison Wesley Publishing Company, 1993.
[14] D. Lin and Z. M. Kedem, ‘Pincer search: A new algorithm for discovering the maximum frequent set’, *Int. Conf. on Extending Database Technology*, (March 1998).
[15] Z. Maazouzi, N. Andrianarivelo, W. Bousdira, and J. Chabin, ‘CDR: A rewriting based tool to design FPLA circuits’, in *International Conference on Artifi cial Intelligence and Symbolic Computation (AISC’2000)*, ed., Springer-Verlag, volume 1930, (2000).
[16] S. Malik, A.R. Wang, R.K. Brayton, and A.L. Sangiovanni-Vincentelli, ‘Logic verifi cation using binary decision diagrams in a logic synthesis environement’, in *Proc. of IEEE/ACM International Conference on Computer-Aided Design*, pp. 6–9, (1988).
[17] S. Minato, ‘Shared binary decision diagram with attributed edges for effi cient boolean function manipulation’, in *Proc. 27th Design Automation Conference*, pp. 52–57, (June 1990).
[18] A. Salleb and C. Vrain, ‘An application of association rules discovery to geographic information systems’, *In 4th European Conference on Principles of Data Mining and Knowledge Discovery PKDD*, 613–618, (September 2000).
[19] J. F. Wakerly, *Digital Design Principles and Practices*, Prentice Hall International Editions, 1991.
[20] I. Wegener, *The Complexity of Boolean Functions*, J. Wiley and Sons, 1987.
[21] M. J. Zaki, ‘Scalable algorithms for association mining’, *IEEE Transactions on Knowledge and Data Engineering*, **12**(3), 372–390, (May/June 2000).

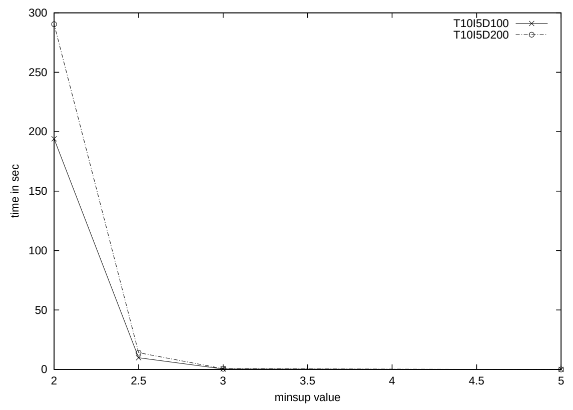

Table 3. T10I5 databases CPU time

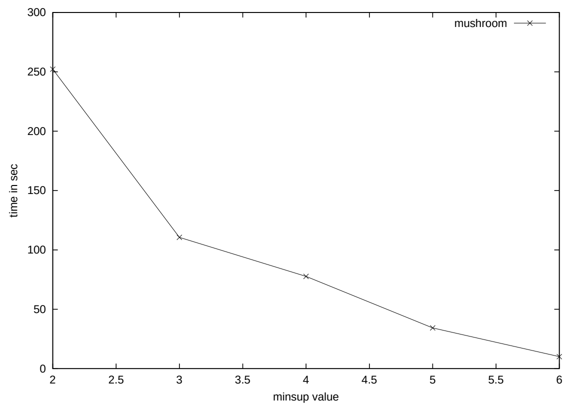

Table 4. Mushroom database CPU time