UNIVERSITE D'ORLEANS
*Faculté des Sciences*
**LIFO**

Laboratoire d'Informatique Fondamentale d'Orléans
4, rue Léonard de Vinci, BP 6759
F-45067 Orléans Cedex 2
FRANCE

Rapport de Recherche

www : http://www.univ-orleans.fr/SCIENCES/LIFO/

# Can We Load Efficiently Dense datasets?

Ansaf Salleb
Christel Vrain

Université d'Orléans, LIFO

Rapport N$^o$ **2004-02**

# Can We Load Efficiently Dense datasets?

Ansaf Salleb Christel Vrain

LIFO, Université d'Orléans, rue Léonard de Vinci
BP 6759, F-45067 Orléans Cedex 2, France
{Salleb, Vrain}@lifo.univ-orleans.fr

We dedicate this work to our colleague Zahir Maazouzi
who collaborated to its realization

**Abstract**

We address the problem of representing and loading transactional datasets relying on a compressed representation of all the transactions of that dataset.
We present an algorithm, named BoolLoader, to load transactional datasets and we give an experimental evaluation on both sparse and dense datasets. It shows that BoolLoader is efficient for loading dense datasets and gives a useful tool to evaluate the density of a dataset.
**Keywords:** Transactional datasets, boolean functions, decision diagrams, dense datasets, frequent itemsets.

## 1 Introduction

*Binary Decision Diagrams (BDDs)* have been successfully used for the representation of boolean functions in various applications of computer science such as very large scale integration (VLSI) computer-aided design (CAD) systems.

In this paper, we would like to share our experience in using BDDs as main data structure for representing and loading transactional datasets.
Basically, in our framework, a dataset is viewed as a vectorial function. For sake of efficiency this function, when possible, is loaded in memory, instead of loading the dataset, by means of BDDs.
We present this new approach and propose an algorithm named BoolLoader, to load transactional datasets. We give also an experimental evaluation on both sparse and dense datasets. It shows that BoolLoader is efficient especially for loading dense datasets which represent a challenging datasets in most data mining tasks and namely in mining frequent itemsets or patterns. Moreover, our contribution can be considered as a tool to evaluate the density of a dataset, which can be very informative on the nature of the dataset before working on it.

The remainder of this paper is organized as follows: In § 2 we give some basic definitions concerning itemsets and transactional datasets. We then show in § 3 how to represent a dataset by a vectorial function. Since we propose to represent a vectorial function by means of a BDD, § 4 is devoted to this data structure, showing its main features. In § 5, we propose an algorithm for moving from a given dataset to a BDD directly. Details on implementation and experimental tests are given and discussed in § 6 and § 7. Computing the frequency of a given itemset is given as an application of our framework in § 8. § 9 presents the state of the art data format and finally, we conclude in § 10 with a summary and a discussion.

## 2 Transactional Datasets

**Definition 1 (Item and itemset)** *An item is an element of a finite set $\mathcal{I} = \{x_1, ..., x_n\}$. A non empty subset of $\mathcal{I}$ is called an itemset.*

**Definition 2 (Transaction)** *We call a* transaction *a subset of $\mathcal{I}$. It is identified by a unique transaction identifier* tid*. In the following, we will denote by $\mathcal{T}$ the set of all transactions identifiers.*

**Definition 3 (Transactional Dataset)** *A transactional Dataset $\mathcal{D}$ is a finite set of distinct pairs, each one constituted by a transaction identifier* tid *and a transaction.*

$$\mathcal{D} = \{(y, X_y) \,/ y \in \mathcal{T},\ X_y \subseteq \mathcal{I}\}$$

| Item | Movie | Producer |
|---|---|---|
| $x_1$ | Harry Potter | C. Columbus |
| $x_2$ | Star Wars II | G. Lucas |
| $x_3$ | Catch me if you can | S. Spielberg |
| $x_4$ | A Beautiful Mind | R. Howard |

| $\mathcal{D}$ | |
|---|---|
| **Tid** | **Transaction** |
| 1 | $x_1, x_2$ |
| 2 | $x_1, x_2, x_4$ |
| 3 | $x_1, x_2$ |
| 4 | $x_3, x_4$ |
| 5 | $x_1, x_2$ |
| 6 | $x_3, x_4$ |
| 7 | $x_1, x_3, x_4$ |
| 8 | $x_1, x_2, x_3$ |
| 9 | $x_1, x_2$ |
| 10 | $x_1, x_3, x_4$ |
| 11 | $x_1, x_2$ |
| 12 | $x_1, x_2, x_3$ |
| 13 | $x_1, x_2$ |
| 14 | $x_1, x_3, x_4$ |
| 15 | $x_3, x_4$ |

$\xrightarrow{\wp}$

| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $f$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 3 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 3 |
| 1 | 1 | 0 | 0 | 6 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 2 |
| 1 | 1 | 1 | 1 | 0 |

Table 1: A transactional dataset and its corresponding truth table

The set of all possible transactions form a lattice $(\mathcal{P}(\mathcal{I}), \subseteq)$ with the minimum $\bot = \emptyset$ and the maximum $\top = \mathcal{I}$.
In the following, we will denote by a *BDT*, a transactional dataset.

**Definition 4 (Frequent itemset)** *The* frequency[1] *of an itemset $X$, is the number of transactions of a BDT, $\mathcal{D}$, containing $X$:*

$$\text{frequency(X)} = |\{(y, X_y) \in \mathcal{D} / X \subseteq X_y\}|$$

*An itemset $X$ is said to be frequent in $\mathcal{D}$ if its frequency is equal or exceeds a given threshold.*

**Example 1** *Let us consider the BDT given in Table 1 as a running example throughout this paper. Let us suppose that it stores recently seen movies by 15 spectators. The dataset $\mathcal{D}$ is defined on the set of items (movies) $\mathcal{I} = \{x_1, x_2, x_3, x_4\}$. The set of tids is given by $\mathcal{T} = \{1, 2, \cdots, 15\}$. Each line in $\mathcal{D}$, gives for a spectator identified by a unique tid, a set of movies. For instance, the spectator 1 has recently seen* Harry Potter *and* Star Wars II. *The itemset $\{x_1, x_3, x_4\}$, noted $x_1x_3x_4$ to simplify, is frequent relatively to the threshold 2 since it appears 3 times in $\mathcal{D}$.*

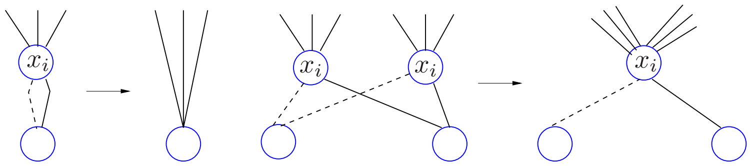
# 3 From Datasets to Vectorial Functions

**Theorem 1 Lattice isomorphism** *The lattice $(\mathcal{P}(\mathcal{I}), \subseteq)$ where $\mathcal{I}$ is a set of $n$ items is isomorphic to the lattice $(\mathbb{B}^n, \leq)$, $\mathbb{B}$=$\{0, 1\}$.*

Indeed, we can define a bijective function $\wp$ as follows:

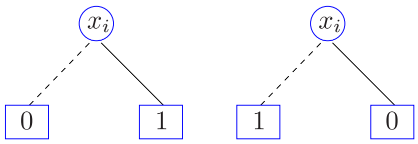
$$\begin{array}{rcl} \wp : \mathcal{P}(\mathcal{I}) & \longrightarrow & \mathbb{B}^n \\ X & \longmapsto & (b_1, b_2, \cdots, b_n) \\ & & \text{where } b_i = 1 \text{ if } \ x_i \ \in \ X, \ 0 \ otherwise \end{array}$$

[1] also called the support when given relatively to $|\mathcal{D}|$




(a) Node elimination          (b) Node sharing

Figure 1: Reduction rules




(a) $x_i$          (b) $\neg x_i$

Figure 2: Trivial BDDs for $x_i$ and $\neg x_i$

Thus, a combination of $n$ items of $\mathcal{I}$ can be represented by a binary vector of $n$-bits, $(b_1, b_2, \cdots, b_n)$, where each bit, $b_i \in \mathbb{B}$, expresses if the corresponding item $x_i$ is included in the transaction or not. We define $Freq(b_1, b_2, \cdots, b_n)$, the number of transactions in $\mathcal{D}$ represented by the vector $(b_1, b_2, \cdots, b_n)$.

Let us consider a truth table $\mathbb{T}^n = [\overrightarrow{e_1}, \ldots, \overrightarrow{e_n}]$, where for each index $j$, $1 \leq j \leq n$, $\overrightarrow{e_j}$ represents the j$^{th}$ vector of $\mathbb{T}^n$, and where $\{\overrightarrow{e_1}, \ldots, \overrightarrow{e_n}\}$ represents a canonical base of $\mathbb{T}^n$. In $\mathbb{T}^n$ each line corresponds to a possible combination, thus a transaction. It is then possible to represent a dataset as a truth table of $n$ variables, $n$ being the number of items of the dataset. We can associate to this truth table a vectorial function $f$, which gives for each line of the truth table (a combination of items), the frequency of that combination in $\mathcal{D}$. Since the structure of the truth table is fixed when $n$ is fixed and variables ordered, the function $f$ is then sufficient to express the entire set of transactions of $\mathcal{D}$.

**Example 2** *The BDT $\mathcal{D}$ of table 1 is represented by $\mathbb{T}^4 = [\overrightarrow{e_1}, \ldots, \overrightarrow{e_4}]$ where:*

$$\overrightarrow{e_1} = 0000\ 0000\ 1111\ 1111, \quad \overrightarrow{e_2} = 0000\ 1111\ 0000\ 1111$$
$$\overrightarrow{e_3} = 0011\ 0011\ 0011\ 0011, \quad \overrightarrow{e_4} = 0101\ 0101\ 0101\ 0101$$

*with the output function* $\overrightarrow{f}$ = 0003 0000 0003 6120.
*For instance, the transaction $\{x_1, x_2, x_3\}$ exists twice in $\mathcal{D}$, it is then represented by the $15^{th}$ line (1110) in the truth table and the value of $f$ is equal to 2. In the same way, the transaction $\{x_1, x_2, x_3, x_4\}$ does not exist in $\mathcal{D}$, it will be represented by the last line (1111) in the truth table with 0 as output function.*

Since $f$ represents a condensed form of a dataset, can we load it into memory instead of loading the dataset itself? moreover how can we represent and handle efficiently a function $f$, knowing that the size of $f$ may be very large? For instance, if we would like to represent a dataset defined on 100 items, we will need to represent a vectorial function of size equal to $2^{100}$, so more than $10^{30}$ unsigned integers! A 'clever' and compact representation has been introduced by Lee [10] and Akers [3], functions are represented by directed acyclic graphs, BDDs, which have several advantages over other approaches to function representations.

# 4 What is a Binary Decision Diagram?

Binary Decision Diagrams (BDDs) are graph representation of boolean functions. A BDD is a directed acyclic graph with 2 terminal nodes 1 and 0. Each non-terminal node has an index to identify a variable of the boolean function, and has also two outgoing edges; the dashed one means that the variable is fixed to 0

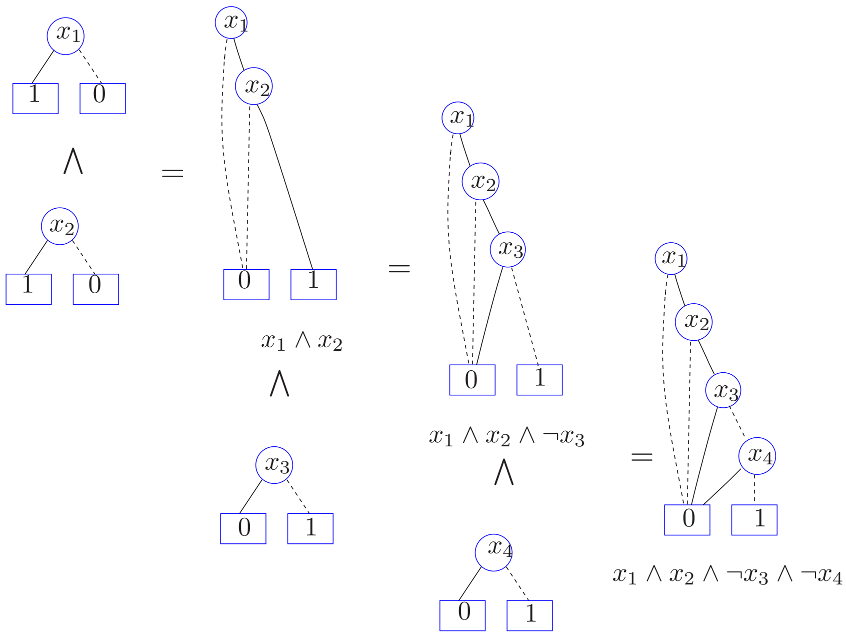



Figure 3: Generation of the BDD of the transaction 1: $x_1 \wedge x_2 \wedge \neg x_3 \wedge \neg x_4$

whereas the edge means that the variable is fixed to 1. An ordered BDD represents a function by a compact and a canonical graph, thanks to the following rules [12] (Figure 1):

1. Choose an order on variables: $x_1 \prec x_2 \prec \ldots \prec x_n$; variables appear in this order in all the paths of the graph and no variable appears more than once in a path.

2. Eliminate all the redundant nodes whose two edges point to the same node.

3. Share all equivalent subgraphs.

Operations on boolean functions represented by BDDs have been defined in [5]. The recursive function *apply* (givel in appendix), given in the appendix, defines basic binary operations between BDDs such as AND ($\wedge$), OR ($\vee$), etc.

A BDD is generated as the results of logic operations. For instance, the BDD of the expression $x_1 \wedge x_2 \wedge \neg x_3 \wedge \neg x_4$ is obtained by first generating trivial BDDs (see figure 2) of $x_1$, $x_2$, $\neg x_3$ and $\neg x_4$, and then by doing the AND operation between these basic BDDs. Let us notice that the trivial BDD of $\neg x_i$ is similar to the BDD of $x_i$ where the two terminal nodes 0 and 1 are exchanged.

An Algebraic Decision Diagram (ADD) [7, 14] is a kind of BDD with integer leaves dedicated to handle functions of boolean variables and integer values, which is the case of the representation presented in § 3.

**Example 3** *Figure 3 shows the successive steps to construct the BDD of the first transaction of the BDT $\mathcal{D}$, $x_1 \wedge x_2$, can be written $x_1 \wedge x_2 \wedge \neg x_3 \wedge \neg x_4$ since the items $x_3$ and $x_4$ are not present in this transaction.*

**Example 4** *Figure 4 shows the BDD associated to the function $f$ of Table 1. For instance, the most right path from the root to the leaf 6 expresses that there are 6 spectators who have seen* Harry Potter *($x_1$) and* Stars Wars *($x_2$) but have not seen the rest of movies ($x_3, x_4$). The leftmost path from the root to the leave 0 expresses that no spectator saw* Stars Wars *without having seen* Harry Potter. *In the following section, we will see how this BDD is constructed.*

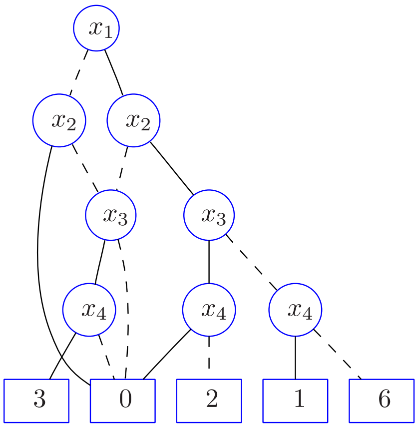



Figure 4: BDD corresponding to the vector $f$

## 5 From Datasets to Decision Diagrams

```
Algorithm BDT2BDD
 Input:   a dataset D
 Output:  a decision diagram BDD_D

BDD_D=NULL
For each transaction t ∈ D
do
   BDD_t=NULL
   For i=1 to n
   do
      if x_i ∈ t then BDD_t=apply(BDD_t, BDD_{x_i}, ∧)
      else BDD_t=apply(BDD_t, BDD_{¬x_i}, ∧)
      fi
   od
   BDD_D=apply(BDD_D, BDD_t, ∨)
od
```

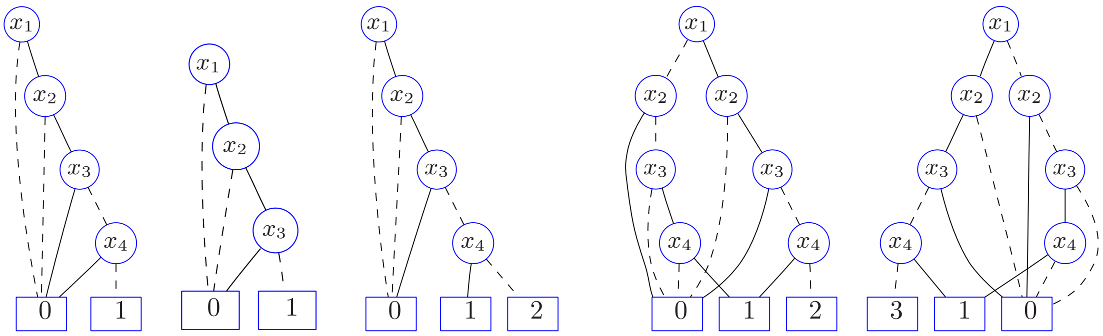
The construction of a BDD representing a dataset is done by scanning only once the dataset. A BDD is constructed for each transaction, and added to the final BDD using the operation $\vee$ between BDDs. Moreover, the reduction rules (Figure 1) are used during the construction process in order to get the more compact possible BDD, by eliminating redundant rules and sharing equivalent subgraphs. We notice here that, **the function $f$ is never computed**; in fact, we move from a transactional dataset to its corresponding BDD directly, *in fine*, this BDD represents actually this function. To resume, the construction of a BDD associated to a BDT, given by the algorithm *BDT2BDD*, is accomplished by considering disjunctions of the transactions which are conjunctions of items (or negation) of items.

**Example 5** *Figure5, represents the evolution of the BDD, while considering transactions one by one, progressively. Figure 5(a) represents the BDD of transaction 1, (b) shows the BDD of the two first transactions, (c) the three first transactions and so on. Finally, the BDD in (f) represents all the transactions of the database.*

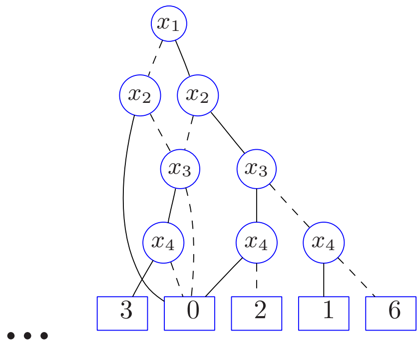
## 6 Implementation

We have developed BoolLoader[2], a prototype in C to load transactional datasets.

[2] Available on `http://www.univ-orleans.fr/SCIENCES/LIFO/Members/salleb/software/BoolLoader`



(a) $t_1$ (b) $t_1 \vee t_2$ (c) $t_1 \vee t_2 \vee t_3$ (d) $t_1 \vee t_2 \vee t_3 \vee t_4$ (e) $t_1 \vee \cdots \vee t_5$



(f) $t_1 \vee \cdots \vee t_{15}$

Figure 5: Example of construction of a reduced BDD of $\mathcal{D}$

We have used BDDs as main data structure to represent datasets, more precisely ADDs for the reason evoked in §4. In order to optimize the memory usage, we have also used shared BDDs [11]. In our implementation, we used the CUDD[3] library. This free library can manage BDDs with a number of nodes which can reach $2^{28}$, *i.e.*, more than 250 million nodes! The nodes managed by CUDD have a size of 16 bytes, one of the smallest sizes of the existing libraries. The maximal number of variables managed by CUDD is equal to $2^{16}$ so 65 536 variables. These characteristics are nice for loading datasets using BDDs.

# 7 Experimental Results

We give the experimental results obtained with BoolLoader on both real and artificial datasets (table 2). Experiments have been performed on a PC Pentium 4, 2.66 GHz processor with 512 Mb of main memory, running under Linux Mandrake 9.2. Artificial datasets have been generated using the *IBM generator*[4]. Concerning real dense datasets we have experimented BoolLoader on some known benchmarks [5], mainly *Mushroom, Chess, Accidents, Retail* and *Connect*.

With our current implementation, we can handle databases of about $D \times N = 10^7$ where $D$ is the number of transactions and $N$ the number of items.

We propose a measure called Sparseness coefficient defined as follows:

$$Sparseness\ coefficient = \frac{\#nodes}{\#transactions \times \#items}\%$$

[3] http://www.bdd-portal.org/cud.html

[4] http://www.almaden.ibm.com/cs/quest/syndata.html

[5] http://fimi.cs.helsinki.fi/data/

| Database | #items | T | #transactions | #nodes | Time(s) | Spars. Coef.(%) |
|---|---|---|---|---|---|---|
| Mushroom | 120 | 23 | 8 124 | 3225 | 1.47 | 0.334 |
| Connect | 130 | 43 | 67 557 | 171 662 | 26.39 | 1.970 |
| Chess | 76 | 37 | 3 196 | 18 553 | 0.54 | 7.741 |
| Accidents | 468 | 33 | 340 183 | 7 238 419 | stopped at 108000 trans. | 14.32 |
| Retail | 16 469 | 10 | 88 162 | 9 818 582 | stopped at 3500 trans. | 17.03 |
| T10I4D10K | 50 | 10 | 9 823 | 102 110 | 1.56 | 20.79 |
| | 100 | 10 | 9 824 | 316 511 | 2.94 | 32.21 |
| | 500 | 10 | 9 853 | 2 557 122 | 25.83 | 51.90 |
| | 1000 | 10 | 9 820 | 5 480 596 | 59.57 | 58.81 |
| T10I4D100K | 50 | 10 | 98 273 | 479 382 | 2.32 | 9.75 |
| | 100 | 10 | 98 394 | 1 985 437 | 55.41 | 20.17 |
| | 150 | 10 | 98 387 | 4 131 643 | 93.04 | 28.18 |
| T10I8D100K | 50 | 10 | 84 824 | 442 671 | 17.15 | 10.43 |
| | 100 | 10 | 84 823 | 1 694 029 | 41.84 | 19.97 |
| | 150 | 10 | 85 027 | 3 228 899 | 68.77 | 25.48 |
| T20I6D100K | 50 | 20 | 99 913 | 916 315 | 27.84 | 18.34 |
| | 100 | 20 | 99 919 | 4 198 947 | 62.52 | 42.02 |
| | 150 | 20 | 99 924 | 7 479 490 | stopped at 95 000 trans. | 52.84 |
| T20I18D100K | 50 | 20 | 81 428 | 632 518 | 19.99 | 15.53 |
| | 100 | 20 | 81 472 | 2 485 199 | 46.10 | 30.50 |
| | 150 | 20 | 81 639 | 4 330 529 | 72.02 | 35.60 |
| T40I10D100K | 50 | 40 | 100 000 | 617 024 | 27.85 | 12.34 |
| | 100 | 40 | 100 000 | 5 561 767 | 72.28 | 55.61 |
| | 150 | 40 | 100 000 | 9 070 580 | stopped at 90 000 trans. | 67.64 |
| T40I35D100K | 50 | 40 | 85 445 | 470 245 | 20.75 | 11.00 |
| | 100 | 40 | 85 410 | 3 175 313 | 50.72 | 37.17 |
| | 150 | 40 | 85 646 | 5 574 661 | 88.04 | 43.68 |

Table 2: Experimental results on real and artificial datasets. For these latters, T denotes the average items per transaction, I the average length of maximal patterns and D the number of transactions. Let us notice that when I is close to T, *i.e.* when the dataset is dense, BoolLoader is more efficient than when I is smaller than T. All times given in our experiments are obtained using the *time* linux command including both system and user time.

It is an idea of the sparseness of the database. Truly, mushroom, connect and chess are known to be dense (with long maximal itemsets) and their spareseness coefficients are less than 10%, whereas, sparse synthetic datasets such as T10I4D10KN1000, have high coefficient.

In order to study the link between spaseness and our coefficient, we have generated several artificial datasets. Experiments have shown that sparseness is linked to the number of transactions and the number of items (figure 6).

Let us notice that some other datasets such as Pumsb, Pumsb*, Accidents seem to be intractable by BoolLoader. However, studying the evolution of the number of nodes according to the number of already processed transactions (figure 7), except for mushroom, we notice a quite linear relationship between these two dimensions. It would allow to extrapolate the number of nodes needed for the loading of the full database. Concerning mushroom, we observe in figure 7(a), that this dataset shows a particular evolution during the loading process. In fact, in mushroom maximal itemsets are long (about 22 items [8]) which is quite the average length of a transaction. This means that in the corresponding BDD, many paths are shared and this explains the few number of nodes and the very low value of the spaseness quoefficient of that dataset.

Let us also notice that as a side effect, our study (figure 6) shows that artificial databases do not behave as real databases (they are too smooth); this point has already been pointed out in [23].

Finally, we mention that in our experiments no preprocessing has been done on the dataset. It could be interesting to find a good ordering of the variables in order to build a more condensed BDD, but we did not do this because it is known to be a NP-complete problem [18, 12] and it would have increased the execution time.

# 8 Application: itemset frequency computation

Given a dataset $\mathcal{D}$ represented by a BDD. How can we compute the frequency of a given itemset $X$ in $\mathcal{D}$? To achieve this task, let us first define an operator *Prod* as follows[6]:

$$\begin{array}{llll} \textit{Prod}: & \mathcal{P}(\mathcal{I}) & \longrightarrow & \mathbb{N}^{2^n} \\ & X & \longmapsto & f_X = f \times e^n_{i_1} \times \cdots \times e^n_{i_k}, \\ & & & \text{where } i_1 \cdots i_k \text{ are indexes of } X \text{ variables} \end{array}$$

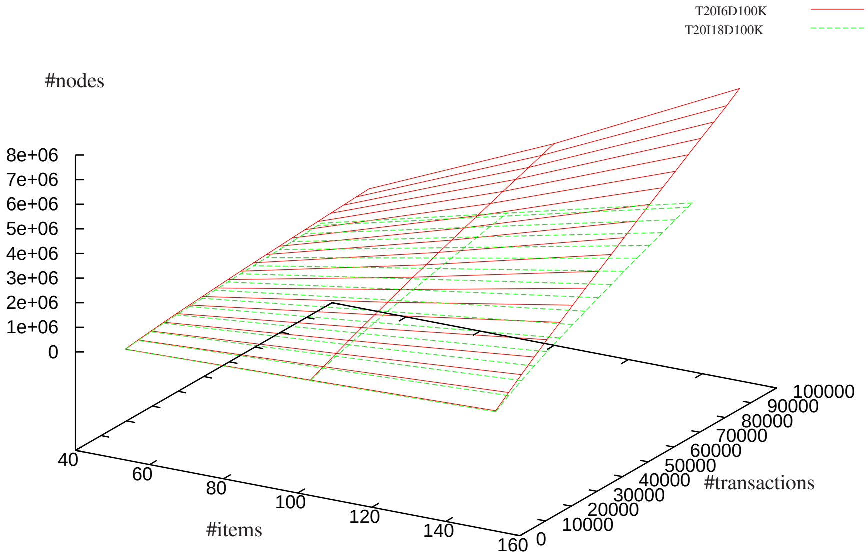
In a certain way, the product 'highlights' some positions in the vectorial function which support the itemsets by switching to zero the remaining positions. Notice that the vector $f_X$ associated to the itemset X can also be represented by a BDD which is actually a sub-BDD of the BDD of $f$. *Prod* is achieved by using operations on BDDs. In order to get the frequency of $X$, one should compute the sum of the elements of its vector $f_X$

**Example 6** *Let us consider the two following itemsets: $x_2$, $x_2x_3\neg x_4$. Intuitively, the last itemset means for example that we are interested in the dataset only by the spectators who have seen* Star Wars II *and* Catch me if you can *but have not seen* A Beautiful Brain*, furthermore we* do not care *if these spectators have watched* Harry Potter *or not. By the second example, we will see how the operator* Prod *can be extended to itemsets* **with negation**.

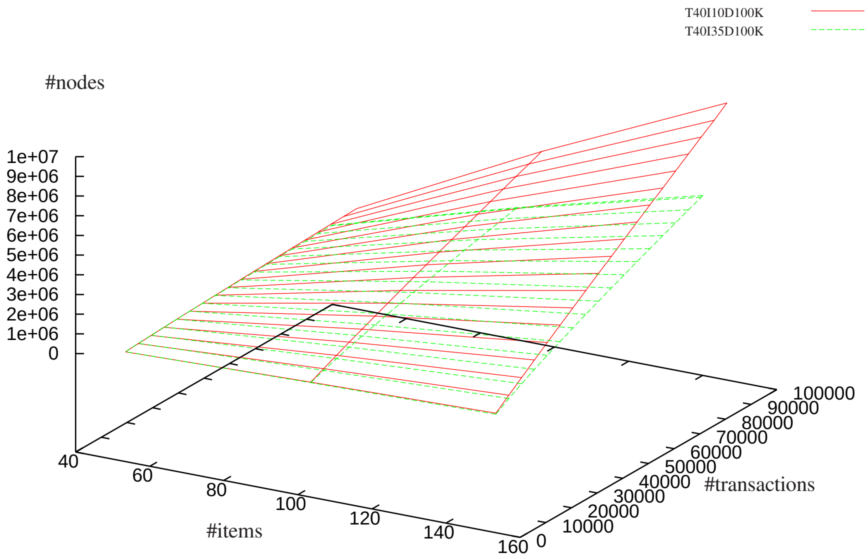
$$\begin{array}{lcl} f_{x_2} & = & f \times e_2 \\ & = & (0003\ 0000\ 0003\ 6120) \times (0000\ 1111\ 0000\ 1111) \\ & = & 0000\ 0000\ 0000\ 6120 \\ f_{x_2x_3\neg x_4} & = & (f \times e_2) \times e_3 \times \neg e_4 = f_{x_2} \times e_3 \times \neg e_4 \\ & = & f_{x_2} \times e_3 \times \neg e_4 \\ & = & (0000\ 0000\ 0000\ 0020) \times (1010\ 1010\ 1010\ 1010) \\ & = & 0000\ 0000\ 0000\ 0020 \end{array}$$

*Figure 8 gives the BDDs corresponding to $f_{x_2}$ and $f_{x_2x_3\neg x_4}$.*
*We have, frequency($x_2$)=9 and frequency($x_2x_3\neg x_4$)=2.*

[6] We note here that the operation $\times$ is not the vectorial product. $(u_1,\ldots,u_{2^n}) \times (v_1,\ldots,v_{2^n}) = (u_1 \cdot v_1,\ldots,u_{2^n} \cdot v_{2^n})$.




(a)




(b)

Figure 6: Evolution of the number of nodes according to the number of transactions and items in artificial datasets

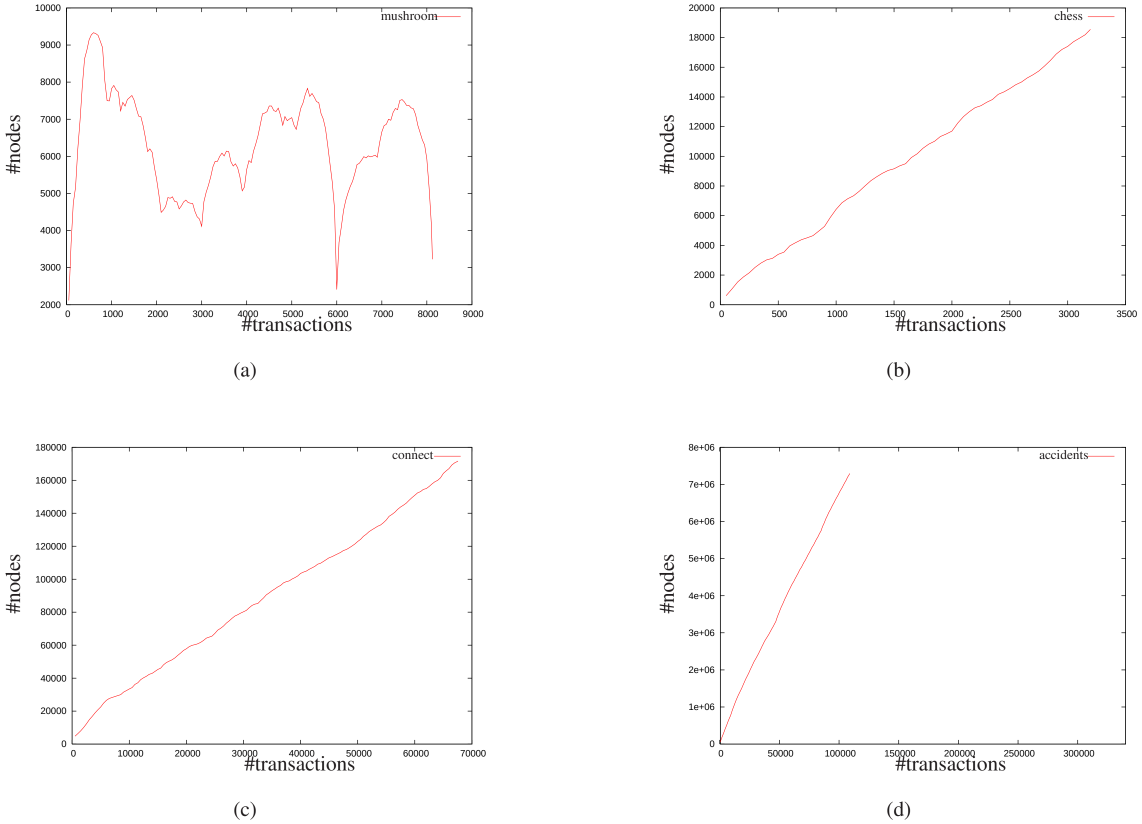



Figure 7: Evolution of the number of nodes according to the number of transactions in real datasets

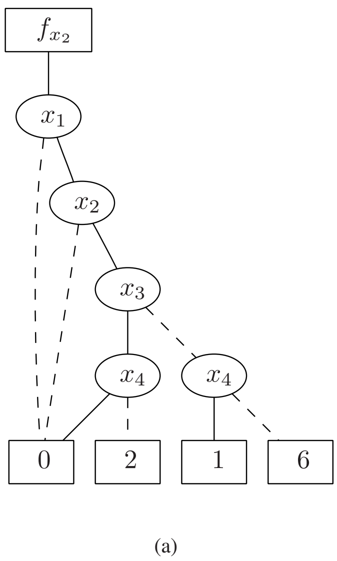



Figure 8: BDDs associated to $x_2$ (a) and $x_2 x_3 \neg x_4$ (b)

# 9 State of the Art Data Formats

In data mining and namely in mining frequents patterns and association rules, four tendencies for representing and handling transactional datasets can be distinguished:

1. **Horizontal format**

   Used in the algorithm *Apriori* [2], this format is considered as the classical and the pioneer representation. The the dataset is seen as a succession of transactions each one identified by a tid. This format is also used for mining maximal frequent itemsets in other algorithms such as *MaxMiner* [4] and *DepthProject* [1].

2. **Vertical format**

   Vertical format used by *Eclat* [22] and *Partition* [16] consists in representing the dataset vertically by giving to each item its *tidset*, *i.e.* the set of transactions containing this items. Another recent vertical format named *Diffset* has been proposed in [20] and in [21]. It consists in keeping only track of differences between tidsets. This format is used in *Charm* [19] for mining closed itemsets.

3. **Bitvectors**

   Bitvectors are used in the algorithm *Mafia* [6] and *Viper* [17]. This format consists in representing data as bitvectors compressed using the Run Length Encoding (RLE).

4. **Fp-tree**

   This data structure is an extended prefix-tree structure for storing compressed and crucial information about frequent patterns. This data structure has been used in *Fp-growth* [9] algorithm for mining frequent patterns but also in *Closet* for mining frequent closed itemsets [13].

# 10 Conclusion

In this paper, we present BoolLoader, a tool for representing a transactional database by a BDD. Our aim when designing that tool was to investigate the use of such data structure for data mining. For the time being, we have studied the feasability of that representation. We have determinded that boolLoader is well suited for some databases and we have given some limits of that approach in terms of the number of items and the number of transactions. Beyond this study, we believe that BoolLoader and the Sparseness Coefficient that we have introduced could be an intersting estimator of the density of the database and thus could be used in a preprocessing task. Moreover, we have shown in [15] how such a data structure could be used in a data mining task, mainly for finding frequent itemsets. In the future, we would like to study other strategies for building the BDD from the transactional database.

We conclude that BDDs may be an interesting technique for solving various problems in machine learning and in data mining.

# References


[1] R. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. Depth first generation of long patterns. In *Intl. Conf. on Knowledge Discovery and Data Mining*, 2000.

[2] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 1994.

[3] S.B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27:509–516, 1978.

[4] R. J. Bayardo. Efficiently mining long patterns from databases. In *Proc. 1998 ACM-SIGMOD Int. Conf. Management of Data*, pages 85–93, Seattle, Washington, June 1998.

[5] R.E. Bryant. Graph-based algorithms for boolean functions manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, August 1986.

[6] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *Proceedings of the 17th International Conference on Data Engineering*, pages 443–452, Heidelberg, Germany, April 2001.

[7] E. Clarke, M. Fujita, P. McGeer, K. McMillan, J. Yang, and X. Zhao. Multi terminal binary decision diagrams: An efficient data structure for matrix representation. In *In Int'l Workshop on Logic Synth.*, pages P6a:1–15, 1993.

[8] K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. In *Proceedings of 1st IEEE International Conference on Data Mining*, San Jose, November 2001.

[9] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. *In ACM-SIGMOD Int. Conf. on Management of Data*, pages 1–12, May 2000.

[10] C.Y. Lee. Representation of Switching Circuits by Binary-Decision Programs. *Bell Systems Technical Journal*, 38:985–999, July 1959.

[11] S. Minato. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *Proc. 27th Design Automation Conference*, pages 52–57, June 1990.

[12] S. Minato. *Binary Decision Diagrams and Applications for VLSI CAD*. Kluwer Academic Publishers, 1996.

[13] Jian Pei, Jiawei Han, and Runying Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 21–30, 2000.

[14] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *IEEE /ACM International Conference on CAD*, pages 188–191, Santa Clara, California, 1993. IEEE Computer Society Press.

[15] A. Salleb, Z. Maazouzi, and C. Vrain. Mining maximal frequent itemsets by a boolean approach. In IOS Press Amsterdam F. van Harmelen, editor, *ECAI*, pages 385–389, Lyon, France, July 21-26 2002. Proceedings of the 15th European Conference on Artificial Intelligence.

[16] A. Savasere, E. Omiecinsky, and S. Navathe. An efficient algorithm for mining association rules in large databases. *In 21st Int'l Conf. on Very Large Databases (VLDB)*, September 1995.

[17] P. Shenoy, J. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, , and D. Shah. Turbo-charging vertical mining of large databases. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 2000.

[18] Tani, Hamaguchi, and Yajima. The complexity of the optimal variable ordering problems of shared binary decision diagrams. In *ISAAC: 4th International Symposium on Algorithms and Computation (formerly SIGAL International Symposium on Algorithms)*, 1993.

[19] M. Zaki and C. Hsiao. Charm: an efficient algorithm for closed association rule mining. In *Proceedings of 2nd SIAM International Conference on Data Mining*, April 2002.

[20] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. In *9th International Conference on Knowledge Discovery and Data Mining*, Washington, DC, August 2003.

[21] Mohammed J. Zaki and Karam Gouda. Fast vertical mining using diffsets. Technical Report 01-1, RPI, 2001.

[22] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. New algorithms for fast discovery of association rules. In David Heckerman, Heikki Mannila, Daryl Pregibon, Ramasamy Uthurusamy, and Menlo Park, editors, *3rd Intl. Conf. on Knowledge Discovery and Data Mining*, pages 283–296. AAAI Press, 1997.

[23] Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In Foster Provost and Ramakrishnan Srikant, editors, *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 401–406, 2001.

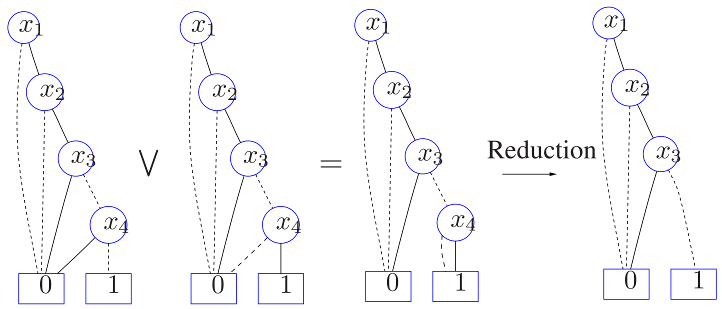



Figure 9: BDD of $t_1 \vee t_2$ using Apply

# Appendix

The function **Apply** given in the following takes two BDDs root$_1$ and root$_2$ and computes the operation Op between them. The function generates the root node of a new BDD corresponding to the product. Let $N_1$ be any node in the first BDD and let $N_2$ be any node in the second one. We can consider these cases: If $N_1$ and $N_2$ are terminal nodes, then we create in the resulting BDD a terminal node with a value equal to $(N_1 \rightarrow$ value $)$ **Op** $(N_2 \rightarrow$ value$)$. Otherwise, if at least one of the two nodes is a non terminal one, but the two nodes have the same index, we create a new node in the result and recall recursively ApplyR on the left part of $N_1$ and the left part of $N_2$ to have the left part of the new node $N$, and recall recursively ApplyR on the right part of $N_1$ and the right part of $N_2$ to have the right part of the new node $N$. If the index of $N_2$ is greater than the index of $N_1$ (including the case when $N_2$ is a terminal node), we create a new Node with the same index as $N_2$ and we apply recursively ApplyR on $left(N_1)$ and $N_2$ to generate $left(N)$ and on $right(N_1)$ and $N_2$ to generate $right(N)$. The reasoning is similar when $N_1$ index is greater than the $N_2$ index. See [5] from which this algorithm is inspired, for more details. See also figure 9 for an example related to the dataset of this paper.

**Function Apply (root$_1$, root$_2$ : BDD; Op) : BDD**

```
//Op is a binary operation (∧, ∨, ···)
```

1. BDD$_{result}$=ApplyR(root$_1$, root$_2$)
2. return Reduction(BDD$_{result}$)

**Function ApplyR ($N_1, N_2$ :BDD): BDD**

1. Create a node $N$
2. if terminal($N_1$) and terminal($N_2$) then
   $N \rightarrow$ value $:= (N_1 \rightarrow$ value$)$ **Op** $(N_2 \rightarrow$ value$)$
   $N \rightarrow$ Son$_0 :=$ NULL
   $N \rightarrow$ Son$_1 :=$ NULL
   $N \rightarrow$ index $:= n$+1
3. else
4. if $N_1 \rightarrow index = N_2 \rightarrow$ index then
   $N \rightarrow index = N_1 \rightarrow$ index
   $N \rightarrow$Son$_0$ :=ApplyR($N_1 \rightarrow$Son$_0, N_2 \rightarrow$Son$_0$)
   $N \rightarrow$Son$_1$ :=ApplyR($N_1 \rightarrow$Son$_1, N_2 \rightarrow$Son$_1$)
5. else
6. if $N_2 \rightarrow$ index $>$ $N_1 \rightarrow$index then
   $N \rightarrow$ index =$N_1 \rightarrow$ index
   $N \rightarrow$ Son$_0 :=$ ApplyR($N_1 \rightarrow$Son$_0, N_2$)
   $N \rightarrow$ Son$_1 :=$ ApplyR($N_1 \rightarrow$Son$_1, N_2$)
7. else
   $N \rightarrow$ index = $N_2 \rightarrow index$
   $N \rightarrow$ Son$_0 :=$ ApplyR($N_1, N_2 \rightarrow$Son$_0$)
   $N \rightarrow$ Son$_1 :=$ ApplyR($N_1, N_2 \rightarrow$Son$_1$)
8. fi
9. fi
10. fi
11. return $N$