Under consideration for publication in Knowledge and Information Systems

PGLCM: Efficient Parallel Mining of Closed Frequent Gradual Itemsets

Trong Dinh Thac Do[1,2], Alexandre Termier[1], Anne Laurent[2], Benjamin Negrevergne[3], Behrooz Omidvar-Tehrani[1], Sihem Amer-Yahia[1]

[1]LIG, CNRS UMR 5217, University of Grenoble, Grenoble, France
[2]LIRMM, CNRS UMR 5506, University of Montpellier II, Montpellier, France
[3]Department of Computer Science, KU Leuven, Belgium

Abstract. Numerical data (e.g., DNA micro-array data, sensor data) pose a challenging problem to existing frequent pattern mining methods which hardly handle them. In this framework, gradual patterns have been recently proposed to extract covariations of attributes, such as: *"When X increases, Y decreases"*. There exist some algorithms for mining frequent gradual patterns, but they cannot scale to real-world databases. We present in this paper GLCM, the first algorithm for mining closed frequent gradual patterns, which proposes strong complexity guarantees: the mining time is linear with the number of closed frequent gradual itemsets. Our experimental study shows that GLCM is two orders of magnitude faster than the state of the art, with a constant low memory usage. We also present PGLCM, a parallelization of GLCM capable of exploiting multicore processors, with good scale-up properties on complex datasets. These algorithms are the first algorithms capable of mining large real world datasets to discover gradual patterns.

Keywords: Data mining; frequent pattern mining; gradual itemsets; parallelism

1. Introduction

Frequent pattern mining is the component of data mining focused on extracting *patterns* that occur frequently in data. These patterns can be seen as abstractions of the contents of large datasets, potentially providing insightful information. Most of the works on frequent pattern mining have focused on categorical data, either mere sets (frequent itemset mining) (Agrawal et Srikant, 1994), or more

Received Nov 01, 2011
Revised Dec 10, 2013
Accepted Mar 15, 2014

complex data having a structure of sequence (Pei et al, 2001), tree (Asai et al, 2002) or graph (Inokuchi et al, 2000). Few of these works have focused on numerical data. There have been some works on quantitative itemset mining, where items can have numerical values (Srikant et Agrawal, 1996). These works focused on discretizing numerical data in order to handle them the same way as categorical data. Thus, only a small part of the information present in the numerical data was exploited. Despite some improvements (Aumann et Lindell, 2003; Washio et al, 2005) over original works on quantitative association rule mining, analyzing numerical data remained marginal in the field of frequent pattern mining.

However, most corporate data and many scientific datasets have many attributes which are numerical: sales numbers, prices, ages, expression level of a gene, quantity of light received by a sensor, etc.

Recently, a new pattern mining field has emerged for analyzing such data: mining of *gradual patterns*. Gradual patterns can be expressed as covariations of several attributes, for example: *"the higher the age, the higher the salary, the lower the free time"*. An algorithm based on Apriori has been proposed (Di Jorio et al, 2009) for mining gradual patterns. This algorithm, as the original Apriori, can mine simple datasets, but it does not scale on large real-world datasets.

Apriori is the pioneer of all algorithms for mining frequent itemsets, however state-of-the-art algorithms considerably outperform it. One of the major steps was the works of Pasquier et al. (Pasquier et al, 1999), which showed that it was sufficient to mine *closed* frequent itemsets, with run time improvements over an order of magnitude in many cases. Later, the FIMI'04 workshop (Goethals, 2003-2004) made a competition between all closed frequent itemset mining algorithms. The winner was the LCM algorithm (Uno et al, 2004).

LCM is based on a theoretical improvement: its authors showed that a closed frequent itemset could be computed by extension of a unique other closed frequent pattern. The closed frequent itemsets are thus the nodes of a covering tree, over which efficient depth first search strategies can be applied, with a very low memory usage.

Our goal is to exploit the principles giving the good performances of LCM in order to compute efficiently gradual itemsets over large real-world databases. The contribution of this paper is fourfold:

– We show that like for itemsets, it is possible to build a covering tree over the search space of closed frequent gradual itemsets.
– We present an algorithm and an implementation for efficiently mining closed frequent gradual itemsets, based on the principle of the LCM algorithm. This algorithm, GLCM, has the same strong complexity guarantees as LCM: its time complexity is linear in the number of closed frequent gradual itemsets, and its memory complexity does not depend on this number. We experimentally show that our algorithm significantly improves the state of the art.
– We also show a simple parallelization of our algorithm, using the Melinda library (Negrevergne et al, 2010). We experimentally show that the parallel version can take advantage of recent multicore processors and handle large real-world datasets.
– Last, we show two real use cases of gradual pattern mining on movie rating data and on financial data, in order to show the insights that can be gained through gradual patterns.

tid	age	salary	loans	cars
t_1	22	2,500	0	3
t_2	35	3,000	2	1
t_3	33	4,700	1	1
t_4	47	3,900	1	2
t_5	53	3,800	3	2

Table 1. Example dataset

It is especially interesting to note that despite LCM proven efficiency, algorithms based on its principle have mostly remained theoretical (for example Arimura and Uno, 2005). By showing how to design and implement an efficient algorithm with this principle, we hope to help the diffusion of the ideas found in LCM, which are still the key to efficient pattern mining algorithms.

The paper is organized as follows: in Section 2, we present in detail the concept of gradual patterns and related works. In Section 3, we recall the base principles of LCM, show how they can be applied to gradual patterns, and present our algorithm and its parallelization. Section 4 presents detailed experiments both about the sequential and the parallel version of our algorithm. We conclude and give directions for future research in Section 5.

2. Gradual itemsets and related works

In this section, we formally define the notion of gradual itemset and their support in data. We also present the notion of *closed* gradual itemest. These definitions are then positioned w.r.t. the state of the art.

2.1. Preliminary Definitions

For the extraction of gradual itemsets, the datasets of interest are sets of *numerical transaction* defined over a schema.

Definition 2.1 (Transaction, Dataset). Given a schema $\mathcal{S} = \{I_1, \ldots, I_n\}$ where the $I_1, ..., I_n$ are attribute names, and a set of transaction identifiers $\mathcal{T} = \{tid_1, ..., tid_m\}$, a transaction is a couple $t_k = (tid_k, \{(I_1, v_1), ..., (I_n, v_n)\})$ associating a transaction identifier $tid_k \in \mathcal{T}$ with a set of attributes-value couples, where each attribute of $\mathcal{S}$ is given a value in $\mathbb{R}$.

A dataset $\mathcal{D}$ is a set of transactions $t_1, ..., t_m$, where transaction t_k has identifier $tid_k \in \mathcal{T}$.

Table 1 shows an example dataset where $\mathcal{S} = \{age, salary, loans, cars\}$ and $\mathcal{T} = \{t_1, ..., t_5\}$ in the classical form of a table, where the transactions are the rows and the attributes are the columns. For sake of readability, we denote by *tid.att* the value for attribute *att* in transaction identified by *tid*. For example $t_4.cars = 2$.

We now define the notions of *gradual item* and *gradual itemset*, which express monotonous variations of values of several attributes.

Definition 2.2 (Gradual item, gradual itemset). A gradual item is a pair (i, v) of an item (attribute) $i \in S$ and a variation $v \in \{\uparrow, \downarrow\}$ where $\uparrow$ stands for a positive (ascending) variation and $\downarrow$ for a negative (descending) variation.

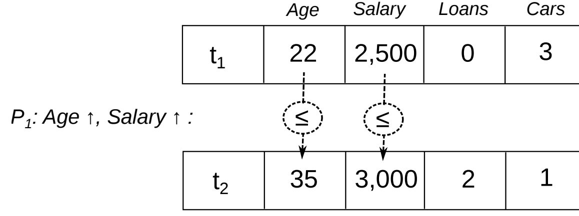


Fig. 1. Exemple of ordering according to a gradual itemset.

A gradual itemset is a non-empty set of gradual items of the form $P = \{(i_{k_1}, v_{k_1}), \ldots, (i_{k_j}, v_{k_j})\}$ where $\{k_1, \ldots, k_j\} \subseteq \{1, \ldots, n\}$ and the $k_1, ..., k_j$ are all distinct.

In our example, the gradual itemset $P_1 = \{(age, \uparrow), (salary, \uparrow)\}$ means *"the higher the age, the higher the salary"*.

Several ways of defining the support of such itemsets in the data have been proposed, that are briefly reviewed at the end of this section. In this paper we use the support definition of (Di Jorio et al, 2009), where the support of a gradual itemset is based on the maximum number of transactions that can be ordered w.r.t. this gradual itemset. In order to explain this definition, the order induced by a gradual itemset is first presented.

Definition 2.3 (Gradual itemset induced order). Let $P = \{(i_{k_1}, v_{k_1}), \ldots, (i_{k_j}, v_{k_j})\}$ be a gradual itemset and $\mathcal{D}$ be a dataset. Two transactions t and t' of $\mathcal{D}$ can be ordered w.r.t. P if all the values of the corresponding items from the gradual itemset can be ordered to respect all the variations of the gradual items of P : for every $l \in [k_1, k_j]$, $t.i_l \leq t'.i_l$ if $v_l =\uparrow$ and $t.i_l \geq t'.i_l$ if $v_l =\downarrow$. The fact that t precedes t' in the order induced by P is denoted $t \triangleleft_P t'$.

For instance, from Table 1, it can be seen that t_1 and t_2 can be ordered with respect to $P_1 = \{(age, \uparrow), (salary, \uparrow)\}$ as $t_1.age \leq t_2.age\ AND\ t_1.salary \leq t_2.salary$: we have $t_1 \triangleleft_{P_1} t_2$. This is illustrated in Figure 1.

This order is only a partial order. For exemple consider t_2 and t_3 of Table 1 : they can't be ordered according to P_1. Clearly, pattern P_1 is not relevant to explain the variations between t_2 and t_3, and more generally all transaction pairs that it can't order. Conversely, a gradual pattern is relevant to explain (part of) the variations occurring in the transactions that it can order. The support definition that we consider below goes further and focuses on the size of the longest **sequences of transactions** that can be ordered according to a gradual itemset. The intuition being that such patterns will be supported by long continuous variations in the data, such continuous variations being particularly desirable to extract in order to better understand the data.

Definition 2.4 (Support of a gradual itemset). Let $L = \langle t_{i_1}, ..., t_{i_p} \rangle$ be a sequence of transactions from $\mathcal{D}$, with $\forall k \in [1..p] \quad i_k \in [1..m]$ and $\forall k, k' \in [1..p]\ k \neq k' \Rightarrow i_k \neq i_{k'}$. Let P be a gradual itemset. L respects P if $\forall k \in [1, p-1]$ we have $t_{i_k} \triangleleft_P t_{i_{k+1}}$. Let $\mathcal{L}_P$ be the set of lists of tuples that respect P.

The formal definition of the support of P is $support(P) = \frac{max_{L \in \mathcal{L}}(|L|)}{|\mathcal{D}|}$, i.e. it is the size of the longest list of tuples that respects P.

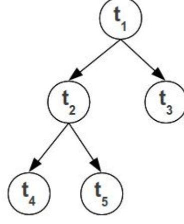


Fig. 2. Partial ordering of transactions of Table 1 according to P_1.

Note that the support of a gradual itemset containing a single gradual item is always 100% as it is always possible to order all the tuples by one column.

As an exemple, consider again the transactions of Table 1 and the pattern $P_1 = \{(age, \uparrow), (salary, \uparrow)\}$. The partial ordering of all transactions according to the order induced by P_1 is represented as a DAG (Directed Acyclic Graph) on Figure 2, where an edge $t_i \rightarrow t_j$ indicates that $t_i \triangleleft_{P_1} t_j$.

The longuest sequences of transactions that can be ordered according to P_1 are $\langle t_1, t_2, t_4 \rangle$ and $\langle t_1, t_2, t_5 \rangle$, both of size 3. Thus $support(P_1) = \frac{3}{5} = 0.6$, meaning that 60% of the input transactions can be ordered consecutively according to P_1.

2.2. Closed Gradual Itemsets

Closed itemsets (Pasquier et al, 1999) have been studied for many years as they represent one of the keys to manage huge databases and to reduce the number of patterns without loss of information. Closure is based on the mathematical notion of Galois Connection. A practical definition is that p is said to be closed if there does not exist any p' such that $p \subset p'$ and $support(p) = support(p')$.

In order to define a Galois closure operator for gradual itemsets, a pair of functions (f, g) defining a *Galois connection* for gradual itemsets have been proposed in (Ayouni et al, 2010).

Function f : Given $\mathcal{L}$ a set of sequences of transactions from $\mathcal{D}$, f returns the gradual itemset P (all the items associated with their respective variations) respecting all transaction sequences in $\mathcal{L}$.

$$f(\mathcal{L}) = P$$

(i) $\mathcal{L}$: A set of sequences of transactions.
(ii) P: a gradual itemset, P is a set of pairs $\{(i_1, v_1), ..., (i_n, v_n)\}$ with $i \in \mathcal{S}$ and $v \in \{\uparrow, \downarrow\}$.
(iii) (i, v) satisfies $\forall L \in \mathcal{L}$, $\forall t_k$, $t_l \in L$ and $k < l$ we have $t_k \triangleleft_P t_l$ if $v =\uparrow$ with $t_l \triangleleft_P t_k$ if $v =\downarrow$.

For example, $f(\{< t_1, t_4, t_5 >, < t_1, t_4, t_2 >\}) = \{(loans, \uparrow), (cars, \downarrow)\}$ because for each item:

- 1(age ↑ or ↓): $t_1 \triangleleft t_4 \triangleleft t_5$ $(22 < 47 < 53)$ but not $t_1 \triangleleft t_4 \triangleleft t_2$ $(47 > 35)$.
- 2(salary ↑ or ↓): neither $t_1 \triangleleft t_4 \triangleleft t_5$ nor $t_5 \triangleleft t_4 \triangleleft t_1$ $(2,500 < 3,900$ but $3,900 > 3,800)$.
- 3(loans ↑): $t_1 \triangleleft t_4 \triangleleft t_5$ $(0 < 1 < 3)$ and $t_1 \triangleleft t_4 \triangleleft t_2$ $(0 < 2 < 3)$.

– 4(cars ↓): $t_5 \triangleleft t_4 \triangleleft t_1$ $(2 = 2 < 3)$ and $t_2 \triangleleft t_4 \triangleleft t_1$ $(1 < 2 < 3)$.

Function g : Given a gradual itemset P, g returns the set of the maximal sequences of transactions $\mathcal{L}$ which respects the variations of all gradual items in P.

$$g(P) = \mathcal{L}$$

(i) P: a gradual itemset, P is a set of pairs $\{(i_1, v_1), ..., (i_n, v_n)\}$ with $i \in \mathcal{S}$ and $v \in \{\uparrow, \downarrow\}$.

(ii) $\mathcal{L}$: A set of maximal sequences of transactions.

(iii) all $L \in \mathcal{L}$ (L has maximal number of tuples) satisfies $\forall t_k,\ t_l \in L,\ k < l$ and $\forall (i, v)$ we have $t_k \triangleleft_P t_l$ if $v =\uparrow$ and $t_l \triangleleft_P t_k$ if $v =\downarrow$.

For example, $g(\{(age, \uparrow), (salary, \uparrow)\}) = \{< t_1, t_2, t_4 >, < t_1, t_2, t_5 >\}$ because with $P = \{(age, \uparrow), (salary, \uparrow)\}$, we have $t_1 \triangleleft_P t_2 \triangleleft_P t_4$ $((22 < 35 < 47)$ for $(age, \uparrow)$, $(2,500 < 3,000 < 3,900)$ for $(salary, \uparrow))$ and $t_1 \triangleleft_P t_2 \triangleleft_P t_5$ $((22 < 35 < 53)$ for $(age, \uparrow)$, $(2,500 < 3,000 < 3,800)$ for $(salary, \uparrow))$ are the two maximal lists of ordered tuples.

Provided these definitions, a gradual itemset p is said to be closed if $f(g(p)) = p$. We define the *closure operator Clo* of a gradual itemset as $Clo(p) = f(g(p))$. By definition $Clo(p)$ is a closed pattern.

Compared to the context of classical items, the main issue here is to manage the fact that g does not return a set of transactions but it returns a *set of sequences of transactions*.

In (Ayouni et al, 2010), these definitions have not been included by the authors within the mining process, but rather as a post-processing step which is not efficient. Indeed, it does not allow to benefit from the runtime and memory reduction and thus does not provide any added value for running the algorithms on huge databases. We thus propose below a novel approach to cope with this.

2.3. Related works

As described above, gradual itemsets (also known as gradual patterns) refer to patterns like *"the higher the age, the higher the salary"*. They can be compared to fuzzy gradual rules that have first been used for command systems some years ago (Dubois et al, 1992; Dubois et al, 1995; Dubois et al, 1996; Dubois et al, 2003), for instance for braking systems: *"the closer the wall, the stronger the brake force"*. Whereas such fuzzy gradual rules are expressed in the same way as the gradual itemsets that we defined, the main difference is that fuzzy gradual rules were not discovered automatically from data. They were designed by human experts and provided as input to expert systems.

Recent works in the pattern mining field have shown that it was feasible to mine automatically such rules from raw data (Berzal et al, 2007; Hüllermeier, 2002; Di Jorio et al, 2009). Many gradual itemsets are output by such methods, so (Ayouni et al, 2010) proposed to mine only closed gradual itemsets in order to reduce the size of the output without loss of information. This preliminary work didn't exploit closure properties to improve the mining algorithm and reduce execution time. However mining gradual itemsets is a costly task in terms of computation time. It was proposed (Laurent et al, 2010) to exploit the parallel

processing capabilities of multicore architectures in order to reduce computation time. The main contribution of the present work is to provide a first algorithm that exploits the properties of closure to reduce the execution time, and that also exploits parallel processing capabilities of multicore processors.

The evaluation of the support of gradual itemsets has been defined in different manners depending on the authors. (Hüllermeier, 2002) is based on regression, while (Berzal et al, 2007) and (Laurent et al, 2009) consider the number of transactions that are concordant and discordant, in the idea of exploiting the Kendall's tau ranking correlation coefficient (Kendall et Babington Smith, 1939). This means that given a gradual itemset P, all pairs of transactions (t_i, t_j) will be compared according to the order induced by P, and the support will be based on the proportion of these pairs that satisfy $t_i \lhd_P t_j$.

In contrast, our definition of support is the one proposed in (Di Jorio et al, 2009) and is based on the length of the longest sequence of transactions that can be ordered *consecutively* according to a gradual itemset P. The interest of this definition is that such sequences of transactions can then be easily presented to the analyst, allowing to isolate and reorder a part of the data and to label it with a description in terms of co-variations (the gradual itemset being this description).

3. Efficiently mining gradual itemsets

In this section, we present our algorithms for mining efficiently closed frequent gradual itemsets. We first explain the principle of the LCM algorithm for mining closed frequent itemsets. We show how we could adapt this algorithm to gradual itemsets with the algorithm GLCM, and we give complexity results on the new algorithm. We then present PGLCM, a parallelization of GLCM, and explain our parallelization strategy.

3.1. LCM principle

LCM is the most efficient algorithm for computing closed frequent itemsets, as shown by the results of the FIMI'04 competition (Goethals, 2003-2004). It is the only such algorithm to exhibit a complexity proven linear with the number of closed frequent itemsets to find, hence its name: ***L**inear time **C**losed itemset **M**iner*. This result comes from an important theoretical advance: the authors of LCM could prove that there existed a covering tree over all the closed frequent itemsets, and the edges of this tree could be computed efficiently at runtime. The closed frequent itemsets can thus be discovered with a depth first algorithm, without maintaining a special storage space for the previously obtained patterns: a closed frequent itemset can be outputted as soon as it is discovered. The only other algorithm adopting a similar approach is DCI-Closed (Lucchese et al, 2004), earlier algorithms had to keep in memory the previously found frequent itemsets, to avoid duplications, which could lead to very important memory usage.

LCM is a depth first search algorithm. Each node of the search tree either corresponds to a closed frequent itemset or to an empty leaf. The pseudo-code of LCM is given in Algorithm 1 (coming from (Uno et al, 2004)).

Each recursive iteration represents a node of the search tree. Its input is a closed frequent pattern. If this pattern is not frequent (line 5), then we are at the

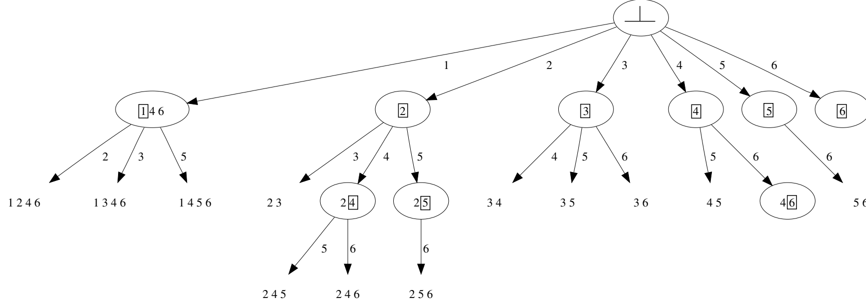


Fig. 3. Example of LCM execution, circled itemsets are the closed frequent itemsets, boxed items are *core_i* values.

Algorithm 1 Algorithm LCM

```
1: Input: T:transaction database, ε:minimum support
2: Output: Enum_ClosedPatterns(⊥) ;

3: Function Enum_ClosedPatterns(P:closed frequent pattern)
4: if P is not frequent then
5:    return ;
6: end if
7: output P ;
8: for i = core_i(P) + 1 to |I| do
9:    Q = Clo(P ∪ i) ;
10:   if P(i-1)=Q(i-1) then
11:      Enum_ClosedPatterns(Q) ;
12:   end if
13: end for
```

end of a branch. Else, the pattern if frequent and the main goal of the iteration is to compute all its direct descendants in the search tree.

This is done with an operation called *prefix preserving extension (ppc-extension)*. Let P be a closed frequent itemset. For an item $i \in P$, we define $P(i) = \{j \mid j \in P \text{ and } j \leq i\}$. Let $core_i(P)$ be the minimum index i such that $\mathcal{T}(P(i)) = \mathcal{T}(P)$ (with $core_i(\perp) = 0$), where $\mathcal{T}(P) = \{t \in \mathcal{T} \mid \ P \subseteq t\}$. Then an itemset Q is called a ppc-extension of P if

(i) $Q = Clo(P \cup \{i\})$ for some $i \in \mathcal{I}$
(ii) $i \notin P$ and $i > core_i(P)$
(iii) $P(i-1) = Q(i-1)$, i.e. P and Q share the same $(i-1)$-prefix.

The authors of LCM have shown that for any closed frequent itemset $Q \neq \perp$, there exist only one closed frequent itemset P such that Q is a ppc-extension of P (Theorem 2 in (Uno et al, 2004)). This is the interest of ppc-extension: it is the very operation that allows the building of a covering tree of closed frequent itemset. Here P will be the unique father of Q, and $\perp$ is the root of the covering tree.

In Algorithm 1, ppc-extension is performed in lines 8-10. For a closed frequent itemset P, all its possible ppc-extensions Q are searched. First, all the i satisfying

condition (ii) are iterated over in line 8. Q is computed according to condition (i) in line 9. And condition (iii) is checked in line 10. In line 11, Q is a ppc-extension of P, so a new recursive iteration (i.e. a new node of the search tree) is built with Q.

Example: Let us consider the following transaction database:

Transaction id	Transaction items
t_1	1,2,3,4,5,6
t_2	2,3,5
t_3	2,5
t_4	1,2,4,5,6
t_5	2,4
t_6	1,4,6
t_7	3,4,6

With $\varepsilon = 3$, the depth first search performed by LCM is shown in Figure 3[1].

By definition $core_i(\bot) = 0$, so any item can be used to extend $\bot$. This is shown by the arcs outgoing from $\bot$, each labelled with the item used for extension. We explain here the branch of item 1. In the case of item 1, $Clo(\{1\}) = \{1, 4, 6\}$, which is our leftmost node for depth 1. It is frequent in the database, so we circle it in the figure as a solution, and the iteration continues. We have to compute $core_i(\{1, 4, 6\})$. Removing 6 from $\{1, 4, 6\}$ gives the itemset $\{1, 4\}$, and $Clo(\{1, 4\}) = \{1, 4, 6\}$, hence $core_i(P) < 4$. Removing 4 and 6 from $\{1, 4, 6\}$ gives the itemset $\{1\}$, and we already know that $Clo(\{1\}) = \{1, 4, 6\}$. 1 is the smallest item i such that $Clo(P(i)) = Clo(P)$, thus $core_i(\{1, 4, 6\}) = 1$. The item 1 is boxed in the figure.

Possible extensions for $\{1, 4, 6\}$ are 2, 3 and 5. The corresponding itemsets are represented below $\{1, 4, 6\}$ in the figure, however they are not frequent so the function immediately returns. The itemsets are not circled in this case, to show that they are not closed frequent itemsets.

3.2. Adapting LCM principle to gradual itemsets

In order to mine closed frequent gradual itemsets with an algorithm similar to LCM, we need to be able to build a covering tree over all the closed frequent gradual itemsets. As seen before, we thus need to redefine *core_i* and ppc-extension for closed frequent gradual itemsets.

Definition 3.1. Let P be a closed gradual itemset, for an item $x \in \mathcal{S}$ we have $P(x) = \{(y, v) \mid (y, v) \in P \text{ and } y \leq x\}$. $core_i(P)$ is the minimum item i such that $(i, v) \in P$ and $g(P(i)) = g(P)$, with $core_i(\bot) = 0$.

Definition 3.2. The closed gradual itemset Q is a ppc-extension of P if

(i) $Q = Clo(P \cup \{(i, v)\})$ for some (i, v), with $i \in \mathcal{I}$ and $v \in \{\uparrow, \downarrow\}$,

(ii) (i, v) satisfies $(i, v) \notin P$ and $(i, \neg v) \notin P$ and $i > core_i(P)$ (with $\neg \uparrow = \downarrow$ and $\neg \downarrow = \uparrow$),

(iii) $P(i - 1) = Q(i - 1)$

[1] The interested reader will have noticed that it is the same example database as in (Uno et al, 2004). However the support value is 3 here instead of 2 in (Uno et al, 2004), hence the difference in output.

Our ppc-extension for gradual itemsets satisfies the following theorem.

Theorem 1. Let $Q \neq \bot$ be a closed gradual itemset.Then, there is just one closed gradual itemset P such that Q is a ppc-extension of P.

To prove the theorem, we state several Lemmas, which are adaptations to gradual itemsets of Lemmas for standard itemsets given in the proof given in (Uno et al, 2004).

From (Di Jorio et al, 2009; Ayouni et al, 2010), the following property holds for closed gradual patterns.

Property 3.1. For any gradual item $(i, v) \notin P$, we have $g(P \cup \{(i, v)\}) = g(P) \cap g(\{(i, v)\})$.

This property is used to state the following lemmas.

Lemma 1. Let P and Q; $P \subseteq Q$ be gradual patterns having $g(P) = g(Q)$. Then, for any gradual item $(i, v) \notin P$; $g(P \cup \{(i, v)\}) = g(Q \cup \{(i, v)\})$.

Proof. $g(P \cup \{(i, v)\}) = g(P) \cap g(\{(i, v)\})$, $g(Q \cup \{(i, v)\}) = g(Q) \cap g(\{(i, v)\})$ (Property 3.1). And $g(P) = g(Q)$. So $g(P \cup \{(i, v)\}) = g(Q \cup \{(i, v)\})$. □

Lemma 2. Let P be a closed gradual pattern and $Q = Clo(P \cup \{(i, v)\})$ be a ppc-extension of P. Then, i is the core index of Q.

Proof. Since by definition $i > core_i(P)$, we have $g(P) = g(P(i))$. From lemma 1, $g(P(i) \cup \{(i, v)\}) = g(P \cup \{(i, v)\})$. Hence by applying f on both side of the equality one have $f(g(P(i) \cup \{(i, v)\})) = f(g(P \cup \{(i, v)\}))$, which can be rewriten as $Clo(P(i) \cup \{(i, v)\}) = Clo(P \cup \{(i, v)\})$. As $Clo(P \cup \{(i, v)\}) = Q$ (by definition of Q) and $Q = Clo(Q)$ (by idempotence of Clo), we can write $Clo(P(i) \cup \{(i, v)\}) = Clo(Q)$.

So $core_i(Q) \leq i$. We have $P(i-1) = Q(i-1) \Rightarrow Clo(Q(i-1)) = Clo(P(i-1)) = P \neq Q$. So $core_i(Q) > i - 1$. Together with $core_i(Q) \leq i$, we have $core_i(Q) = i$. □

Lemma 3. Let $Q \neq \bot$ be a closed gradual pattern, and $P = Clo(Q(core_i(Q) - 1))$. Then, Q is a ppc-extension of P.

Proof. Since $g(P) = g(Q(core_i(Q) - 1))$, for some (i, v) with $i \in \mathcal{I}$ and $v \in \{\uparrow, \downarrow\}$ we have $g(P \cup (i, v)) = g(Q(core_i(Q) - 1) \cup (i, v)) = g(Q(core_i(Q)))$. This implies $Q = Clo(P \cup \{(i, v)\}) \Rightarrow$ condition (i) of ppc-extension is satisfied. $P = Clo(Q(core_i(Q) - 1)) \Rightarrow core_i(P) \leq i - 1$. Thus, Q satisfies condition (ii) of ppc-extension. Since $P \subset Q$ and $Q(i-1) \subseteq P$, we have $P(i-1) = Q(i-1)$. Thus, Q satisfies condition (iii) of ppc-extension. □

Proof of Theorem 1

Proof. Assume that P is a a closed gradual pattern, and Q is a ppc-extension of P. Let $P = Clo(Q(core_i(Q) - 1))$. P' is another closed gradual pattern, $P' \neq P$ and Q is also a ppc-extension of P'. From condition (i) of ppc-extension, $Q = Clo(P' \cup \{(i, v)\})$. And from lemma 2, $i = core_i(Q)$. From condition (iii) of ppc-extension, $P'(i-1) = Q(i-1) = P(i-1)$. Hence $Clo(P'(i-1)) = Clo(Q(i-1)) = P \neq P'$, and $core_i(P') \geq i$. This violates condition (ii) of ppc-extension, and is a contradiction. □

3.3. The GLCM algorithm

Thanks to the ppc-extension for gradual itemsets, we can write a mining algorithm similar to LCM. We present our GLCM algorithm in Algorithm 2.

For simplicity (and performance) reasons, we have decided to represent gradual items as integers, instead of pairs. A gradual item (i, v) will be encoded by the integer $enc(i, v)$ such as:

$$enc(i,v) = \begin{cases} 2i & \text{if } (i,v) = (i,\uparrow) \\ 2i+1 & \text{if } (i,v) = (i,\downarrow) \end{cases}$$

The even integers representing positive (ascending) variations and the odd integers representing negative (descending) variations. With this encoding, we force the items of $\mathcal{I}$ to be $[0, n-1]$, with a renaming if necessary.

With such a coding of gradual items, ppc-extension for gradual itemsets is even closer to ppc-extension for itemsets: point (i) and (iii) are strictly identical, where i stands for the encoding of a gradual item and $core_i$ is applied on encoded gradual itemsets and returns itself an integer code instead of an item. To avoid confusion with item 0, we fix $core_i(\perp) = -1$. For point (ii), we have to check the (encoded) value of $core_i$. If it is an even value of the form $enc(i, v) = 2i$, it means that the gradual itemset P contains $(i, \uparrow)$. It would not make sense to try to extend P with $enc(i, v) + 1 = 2i + 1$, corresponding to the gradual item $(i, \downarrow)$. In this case we directly skip to $enc(i, v) + 2$, i.e. $(i + 1, \uparrow)$. This verification is handled in lines 18-23 of Algorithm 2. Lines 23-27 show the ppc-extension itself, and are very similar to LCM.

The differences between LCM and GLCM lie first in a small optimization specific to gradual itemset in lines 14-15. Any gradual itemset, representing co-variations items, has a symmetric gradual itemset where the items are the same and the variations are all reversed. For example, the symmetric of $\{(1,\uparrow), (2,\downarrow), (3,\uparrow)\}$ is $\{(1,\downarrow), (2,\uparrow), (3,\downarrow)\}$, supported by the same tid sequences but in the reverse order. It is redundant to compute a gradual itemset and its opposite, so we arbitrarily decided to compute only gradual itemsets whose first variation is ascending, they represent themselves and their opposite as well.

The computation of support and of closure rely on an efficient exploitation of binary matrices, they are thus explained in details in the next subsection.

3.4. Computation of Support and Closure with Binary Matrices

Computing support and closure with the support definition given in Section 2, are complex operations, that need to be performed as efficiently as possible. To perform these operations we exploit the method introduced in (Di Jorio et al, 2009), which is based on binary matrices and can be implemented very efficiently. One binary matrix is contructed per gradual itemset P. It represents the adjacency matrix of a graph where the nodes are the transactions, and where there is an edge between nodes n_a and n_b representing transactions t_a and t_b iff the ordering $t_a \lhd_P t_b$ holds. Such matrix thus represents all the transaction sequences for the order induced by the gradual itemset P, which are necessary for computing support and closure of P.

Support computation: Computing the support of a gradual itemsets boils down to : 1) compute the graph of the order induced over transactions by the

Algorithm 2 Algorithm GLCM

Require: $\mathcal{T}$: transaction database, ε: minimum support
Ensure: All closed frequent gradual itemsets are output

1: **for all** gradual item $(i, v) \in \mathcal{I} \times \{\uparrow, \downarrow\}$ **do**
2: $\quad L_{enc(i,v)} \leftarrow$ tid sorted in v order of item i value
3: $\quad B_{enc(i,v)} \leftarrow$ bitmap matrix associated to $L_{enc(i,v)}$
4: **end for**
5: $\mathcal{B} \leftarrow \{B_1, ..., B_{2\times|\mathcal{I}|}\}$
6: **for all** gradual item encoding $e \in [0, 2 \times |\mathcal{I}| - 1]$ **do**
7: $\quad GlcmLoop(\{e\}, \mathcal{T}, \mathcal{B}, \varepsilon)$
8: **end for**

9: **Function GlcmLoop(**P**:closed gradual itemset,** $\mathcal{T}, \mathcal{B}, \varepsilon$**)**
10: **if** $computeSupport(P, \mathcal{T}, \mathcal{B}) < \varepsilon$ **then**
11: $\quad$ **return** ;
12: **end if**
13: **if** $P[0]$ is odd **then**
14: $\quad$ **return** ; *// Symmetrical itemset has already been tested*
15: **end if**
16: output P ;
17: **if** ($core_i(P)$ is odd) **or** ($core_i(P) = -1$) **then**
18: $\quad k = core_i(P)$
19: **else**
20: $\quad k = core_i(P) + 1$
21: **end if**
22: **for** $e = k + 1$ to $2 \times |\mathcal{I}| - 1$ **do**
23: $\quad Q = Clo(P \cup \{e\}, \mathcal{B})$;
24: $\quad$ **if** $P(e-1) = Q(e-1)$ **then**
25: $\quad\quad$ GlcmLoop($Q, \mathcal{T}, \mathcal{B}, \varepsilon$) ;
26: $\quad$ **end if**
27: **end for**

gradual itemset ; 2) compute the length of the longest path in this graph. Both operations can be done conveniently by using binary matrices.

The adjacency matrix of the graph is explicitely constructed. It exploits the property that performing a logical AND between the binary adjacency matrices of two gradual itemsets gives the binary adjacency matrix of the union of these gradual itemsets.

Exemple : Consider the example dataset of Table 1. The binary adjacency matrices of the 1-gradual itemsets $P_{11} = \{(age, \uparrow)\}$ and $P_{12} = \{(salary, \uparrow)\}$ are given as $BM_{P_{11}}$ in Table 2 and $BM_{P_{12}}$ in 3. One can easily verify that $BM_{P_1} = BM_{P_{11}} AND BM_{P_{12}}$ where BM_{P_1} is the binary adjacency matrix of pattern $P_1 = \{(age, \uparrow), (salary, \uparrow)\}$, shown in Figure 4(a), represented in graphical forma in Figure 4(b).

In this matrix, it can be seen that:

- t_1 precedes t_2, t_3, t_4 and t_5.
- t_2 precedes t_4 and t_5.

	t_1	t_2	t_3	t_4	t_5
t_1	1	1	1	1	1
t_2	0	1	0	1	1
t_3	0	1	1	1	1
t_4	0	0	0	1	1
t_5	0	0	0	0	1

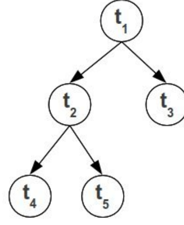
Table 2. Binary matrix corresponding to $\{(age, \uparrow)\}$

	t_1	t_2	t_3	t_4	t_5
t_1	1	1	1	1	1
t_2	0	1	1	1	1
t_3	0	0	1	0	0
t_4	0	0	1	1	0
t_5	0	0	1	1	1

Table 3. Binary matrix corresponding to $\{(salary, \uparrow)\}$

	t_1	t_2	t_3	t_4	t_5
t_1	1	1	1	1	1
t_2	0	1	0	1	1
t_3	0	0	1	0	0
t_4	0	0	0	1	0
t_5	0	0	0	0	1

(a)



(b)

Fig. 4. Binary matrix(a) and graph(b) corresponding to $\{(age, \uparrow), (salary, \uparrow)\}$

This gives $support(\{(age, \uparrow), (salary, \uparrow)\}) = 3/5$ with the longest path $< t_1, t_2, t_4 >$ and $< t_1, t_2, t_5 >$.

In the pseudo-code of Algorithm 2, all the binary matrices of 1-gradual itemsets are computed inside the first loop line 4 and stored in $\mathcal{B}$ in line 6.

The computation of the support itself is a computation of longest path in the graph of the order induced by the gradual itemset. In general this problem is NP-hard, but here as the graph represents a partial order, it is by definition a DAG (directed acyclic graph) so the longuest path can be found in linear time by using a topological sort algorithm. This is the purpose of function *computeSupport*, line 11 of Algorithm 2.

The pseudo code of *computeSupport* is shown in Algorithm 3, it has been first presented in (Di Jorio et al, 2009). The first step of the algorithm is to compute the binary matrix of the order induced by the gradual pattern P, this is done in lines 1-4 by ANDing all the binary matrices of the gradual items of P, that are stored in set $\mathcal{B}$. Then a topological sort is performed, is specificity being that to each vertex of the graph (i.e. each transaction), the algorithm computes the longest path from this vertex to a descendant leaf in the sorted DAG and stores it in the table $FreMap$. The table is initialized in lines 6-8, and the recursive computation of node depths is performed in lines 9-11 by calling for each transaction the recursive function *computeSupportLoop*. *computeSupportLoop* traverses the graph and gives a longest path size of 1 to the leaves. For the internal nodes, their longest path size is the longest path size of their child having the longest path, plus one.

	t_1	t_2	t_3	t_4	t_5	t_6	t_7
t_1	1	1	1	1	1	0	1
t_2	0	1	0	1	1	0	0
t_3	0	0	1	1	1	0	1
t_4	0	0	0	1	1	0	0
t_5	0	0	0	0	1	0	0
t_6	0	0	0	1	1	1	0
t_7	0	0	0	0	0	0	1

Table 4. Binary matrix corresponding to the set of transaction sequences in Figure 5.

Once $FreMap$ is complete, the support is simply the maximal value in $FreMap$.

Algorithm 3 Algorithm *computeSupport*

Require: Gradual itemset P, Transaction database $\mathcal{T}$, Set of binary matrices of gradual items $\mathcal{B}$.
Ensure: The support of P is returned.

1: $BM \leftarrow B_{P[0]}$
2: **for all** gradual item encoding $e \in P$ **do**
3:     $BM \ \& = B_e$
4: **end for**
5: $FreMap$: list of all transactions in $\mathcal{T}$ together with their support.
6: **for all** transaction t_i in $\mathcal{T}$ **do**
7:     $FreMap[t_i] = -1$
8: **end for**
9: **for all** transaction t in $\mathcal{T}$ **do**
10:     $ComputeSupportLoop(t, BM, FreMap)$
11: **end for**
12: **return** $max(FreMap)$

13: **Function computeSupportLoop(t: a transaction in $\mathcal{T}$, BM, $FreMap$)**

14: $Children \leftarrow getChildren(t, BM)$
15: **if** $|Children| = 0$ **then**
16:     $FreMap[t] = 1$; // *t is a leaf*
17: **else**
18:     **for all** $t_i \in Children$ **do**
19:         **if** $FreMap[t_i] = -1$ **then**
20:             $computeSupportLoop(t_i, BM, FreMap)$
21:         **end if**
22:         $FreMap[t] = max(FreMap[t], FreMap[t_i] + 1)$
23:     **end for**
24: **end if**

For example, we have the list of transaction sequences represented by the graph in Figure 5 and the binary matrix representing in Table 4 (corresponding to BM in the code). The $FreMap$ is initialized with seven transaction together with the support -1. We execute the algorithm with all transactions and begin

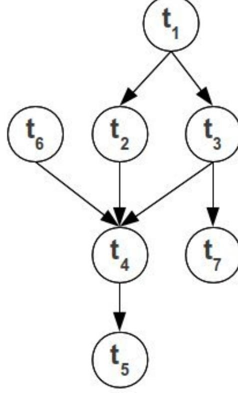


Fig. 5. An example of set of transaction sequences.

(a)

t_1	t_2	t_3	t_4	t_5	t_6	t_7
4	3	3	2	1	-1	1

(b)

t_1	t_2	t_3	t_4	t_5	t_6	t_7
4	3	3	2	1	3	1

Fig. 6. The list of transactions together with their support.

with t_1. Then the algorithm calls itself recursively on all children of t_1 (t_2 and t_3). It continues until we meet the first "leaf" (t_5). We assign $FreMap[t_5] = 1$ and $FreMap[t_4]$ is updated to 2. After executing the $ComputeSupportLoop$ for t_1, we have the $FreMap$ in Figure 6(a). And the final result is shown in Figure 6(b), from which the the support value of 4 is deduced.

Closure computation:

The computation of closure, presented in Algorithm 4, exploits further the properties of binary matrices explained above. The Clo function just consists in composing f and g function, as explained in Section 2. The g function has to return all the (maximal) sequences of transactions supporting input gradual itemset P. In line 13-16 we just compute the binary matrix BM corresponding to the order induced by P, in the same way as for support computation: this matrix contains the longest transaction sequences. Note that to ease understanding of support computation and closure computation independently, the code to compute BM is duplicated in Algorithm 4 and Algorithm 3. In practice, our implementation does this computation only once, during support computation.

To get the actual longest sequences that are the expected result of g, we would have to find all the longest pathes in the graph represented by BM. However as we are only interested in the result of Clo, this intermediary computation can be avoided with the way we compute f.

f takes a set of sequences of transactions, here encoded in a binary matrix BM. It has to check for each item of $\mathcal{I} \times \{\uparrow, \downarrow\}$ if the variations of this item are compatible with each of the input sequences. This comes to check if all the 1 in the input matrix BM can be found in the matrix of gradual item i, B_i (remember

Dataset	Percentage of equal values
C1000A20 (synthetic)	8.8 %
C500A50 (synthetic)	0.24 %
Microarray (real, biology)	8.75 %
Stock (real, finance)	12.4 %

Table 5. Equal values in our datasets

that if $B[x, y] = 1$, it means that $t_x \triangleleft t_y$ for the gradual itemset associated to B). We thus AND BM and B_i for each gradual item i, and keep only the gradual items i such that BM AND $B_i = BM$.

Algorithm 4 Functions Clo and G

Require: Gradual pattern P, Set of binary matrices of gradual items $\mathcal{B}$
Ensure: Returns the closure of P

1: **Function Clo(**$P, \mathcal{B}$**)**
2: **return** $F(G(P, \mathcal{B}), \mathcal{B})$;

3: **Function F(**$BM, \mathcal{B}$**)**
4: $P \leftarrow \emptyset$
5: **for all** gradual item encoding $e \in [1, 2 \times |\mathcal{I}| - 1]$ **do**
6: $tmp \leftarrow BM\&B_e$
7: **if** $tmp = BM$ **then**
8: $P \cup = \{e\}$
9: **end if**
10: **end for**
11: **return** P ;

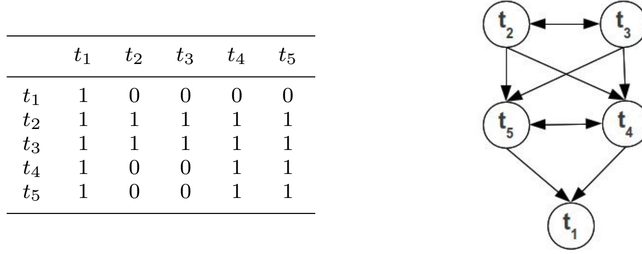
12: **Function G(**$P, \mathcal{B}$**)**
13: $BM \leftarrow B_{P[0]}$
14: **for all** gradual item encoding $e \in P$ **do**
15: $BM \ \& = B_e$
16: **end for**
17: **return** BM ;

The use of binary matrices has the advantage to avoid costly computations with sequence and graph structures, and lead to compact structures thanks to bitmap representation.

Problem of equal values:

In real data, values are not always strictly increasing or strictly decreasing. There can be cases where several transactions have the same value for an attribute. Consider Table 5, where we compute for four datasets (two synthetic, two real-world, presented in details in Section 4), the percentage of values in data that are equal for the same attribute across several transactions.

One can notice that this situation is quite common, especially in real datasets. The problem with equal values is that if two transaction have the same value for an attribute $i \in \mathcal{I}$, the means that at least for gradual itemsets $(i, \uparrow)$ and $(i, \downarrow)$, both $t_x \triangleleft t_y$ and $t_y \triangleleft t_x$ hold. We thus have a cycle in the graph of the order induced by these gradual itemsets, and possibly of some of their super-itemsets.



	t_1	t_2	t_3	t_4	t_5
t_1	1	0	0	0	0
t_2	1	1	1	1	1
t_3	1	1	1	1	1
t_4	1	0	0	1	1
t_5	1	0	0	1	1

Table 6. An example of equal values problem, gradual pattern: $\{(car, \uparrow)\}$.

Algorithm 3 for support computation cannot handle such situation as it expects a DAG.

For instance, in Table 1, both t_4 and t_5 have 2 cars and both t_2 and t_3 have 1 cars. The graph for the order induced by gradual pattern $\{(car, \uparrow)\}$ and its binary matrix are show in Table 6.

Our solution to this problem is simple: for support computation only (i.e. when constructing BM in Algorithm 6), if two transactions t_x and t_y have equal values for all attributes of the gradual itemset, then an order between t_x and t_y is arbitrarily chosen.

This solution does not alter the soundness of result:

- If both t_x and t_y are not included in any longest path, they have no influence on the final result
- If both t_x and t_y are included in a longest path, $t_1, ..., t_x, t_y, ..., t_n$, by assuming $t_x \lhd t_y$, the support is unchanged and remains the number of transactions in the longest path (n).
- It can not be the case that t_x is included in the longest path, $t_1, ..., t_x, ..., t_n$ but t_y is not because all values of t_x and t_y are equal for all attributes of the gradual pattern.

In the example of Table 1, both t_4 and t_5 have two cars and both t_2 and t_3 have one car. We assume that $t_4 \lhd t_5$ (but not $t_5 \lhd t_4$) and $t_2 \lhd t_3$ (but not $t_3 \lhd t_2$). The graph and corresponding binary matrix are shown in Table 7, the support is 5 as expected.

There are no problem for closure computation with equal values, as the algorithm does not assume any particular graph structure.

3.5. Complexity

The GLCM algorithm makes a depth first exploration of a covering tree over all the closed frequent gradual itemsets. Like LCM, it's complexity is thus linear in the number of closed frequent gradual itemsets to find.

For each closed frequent gradual itemset, the complexity of support computation comes to find the longest path in a graph, which as been proven to be linear in the number of nodes of the graph for directed acyclic graphs, i.e. $O(||\mathcal{T}||)$ in our case. The complexity of closed frequent itemset computation is thus, like LCM, dominated by closure computation. Analyzing Algorithm 4 shows that f

	t_1	t_2	t_3	t_4	t_5
t_1	1	0	0	0	0
t_2	1	1	1	1	1
t_3	1	0	1	1	1
t_4	1	0	0	1	1
t_5	1	0	0	0	1

Table 7. Solution for Equal Value Problem.

and g both loop on the items, and make binary matrix computations inside the loops. The time complexity of Clo is thus $O(|\mathcal{I}| \times ||\mathcal{T}||^2)$. Closure operation is embedded in a loop on items in GlcmLoop, so the overall time complexity per closed frequent gradual itemset is $O(|\mathcal{I}|^2 \times ||\mathcal{T}||^2)$. The space complexity mainly depends on the storage of the initial database and of the binary matrices for items, this gives a space complexity of $O(||\mathcal{T}|| + |\mathcal{I}| \times ||\mathcal{T}||^2)$.

GLCM inherits from LCM its good complexity properties. Especially, its space complexity do not depend on the number of closed frequent gradual itemsets to find. This allow in practice to run with a very low and near constant memory usage, whereas other algorithms can use exponentially more memory.

3.6. Parallelization

The existing work for discovering frequent gradual itemsets, Grite (Di Jorio et al, 2009), has been parallelized as the Grite-MT algorithm in order to exploit multicore processors, with good results (Laurent et al, 2010).

We also give a simple parallelization for the GLCM algorithm in this paper, which is based on the works for parallelizing LCM (Negrevergne et al, 2010). The authors have defined a parallelism environment, Melinda, that simplifies the parallelization of existing sequential algorithms by relieving the algorithm designer of the burden of manual thread synchronization while being efficient for parallelizing recursive algorithms.

Melinda is based on the Linda approach (Gelernter, 1989). It consists of a shared memory space, called *TupleSpace*. All the threads can access the TupleSpace, and either deposit or retrieve a data unit called *Tuple*, via the two primitives $get(Tuple)$ and $put(Tuple)$. All the synchronizations for accessing the TupleSpace are handled by Melinda. The main difference with the original Linda approach is that whereas Linda accepted tuples of heterogenous structure and allowed retrieval of tuples through complex queries on tuple values, Melinda is specialized for fixed-structure tuples and does not allow querying given ranges of tuples. These restrictions make Melinda a very low overhead framework than can handle millions of tuples, with a high throughput on tuple insertion/suppression.

Such properties are desirable for the parallelization strategy that we exploit for PGLCM, which is the same as the one we used in previous works for PLCM.

Our observation is that as the enumeration tree of a frequent pattern mining algorithm is difficult to predict beforehand, hence the task decomposition must be as dynamic as possible to avoid load unbalance issues. We consider that an elementary task is the execution of the function *PGlcmLoop* for a pattern given in input, without the recursive calls it generates. This decomposition is fine grained enough to avoid any load unbalance issues in all the experiments that we conducted, that are reported in Section 4.

Algorithm 5 Algorithm PGLCM

Require: $\mathcal{T}$: transaction database, ε: minimum supportt, N: number of threads, $maxD$: depth threshold
Ensure: All closed frequent gradual itemsets are output

1: **for all** gradual item $(i, v) \in \mathcal{I} \times \{\uparrow, \downarrow\}$ **do**
2: $L_{enc(i,v)} \leftarrow$ tid sorted in v order of item i value
3: $B_{enc(i,v)} \leftarrow$ bitmap matrix associated to $L_{enc(i,v)}$
4: **end for**
5: $\mathcal{B} \leftarrow \{B_1, ..., B_{2\times|\mathcal{I}|}\}$
6: **for all** gradual item encoding $e \in [0, 2 \times |\mathcal{I}| - 1]$ **do**
7: $put((\{e\}), 1)$
8: **end for**
9: **wait** for all threads to complete

10: **Function PGlcmLoop(P:closed gradual itemset), d: depth**
11: **if** computeSupport$(P, \mathcal{T}, \mathcal{B}) < \varepsilon$ **then**
12: **return** ;
13: **end if**
14: **if** $P[0]$ is odd **then**
15: **return** ; *// Symmetrical itemset has already been tested*
16: **end if**
17: output P ;
18: **if** ($core_i(P)$ is odd) **or** ($core_i(P) = -1$) **then**
19: $k = core_i(P)$
20: **else**
21: $k = core_i(P) + 1$
22: **end if**
23: **for** $e = k + 1$ to $2 \times |\mathcal{I}| - 1$ **do**
24: $Q = Clo(P \cup \{e\}, \mathcal{B})$;
25: **if** $P(e-1) = Q(e-1)$ **then**
26: **if** $d \leq maxD$ **then**
27: $put((Q, d+1))$; *// Process children in parallel*
28: **else**
29: GlcmLoop$(Q, \mathcal{T}, \mathcal{B}, \varepsilon)$ *// Switch to sequential execution*
30: **end if**
31: **end if**
32: **end for**

The corresponding implementation makes the algorithm PGLCM, whose pseudo-code is shown in Algorithm 5 The main thread first performs the necessary initializations in lines 1-4, as in GLCM, then for each gradual item it

Algorithm 6 Function threadFunction()

1: **while** $get(tuple)$ **do**
2: $P \leftarrow tuple.pattern$;
3: $d \leftarrow tuple.depth$
4: PGlcmLoop((P, d)) ;
5: **end while**

pushes a tuple to the TupleSpace with *put* in lines 6-8. Worker threads execute Algorithm 6 : when idle, they ask a tuple to the TupleSpace with *get*. Then they extract the pattern from this tuple and call the function *PGlcmLoop* with this pattern.

After initialization, some of the individual gradual items will thus be assigned to threads for processing (or all of them if $|\mathcal{I}| \leq numThreads$). *PGlcmLoop* does the computation in the same way as *GlcmLoop* presented in Algorithm 2. Note that the parameters $\mathcal{T}, \mathcal{B}$ and ε of *GlcmLoop* are also given to *PGlcmLoop*. They are constant for all threads, thus have not been shown in Algorithm 5 to improve readability.

In line 27, instead of calling itself recursively, *PGlcmLoop* pushes the patterns it computed into the TupleSpace. New tasks are thus available in the TupleSpace to feed idle threads, allowing the complete computation of the enumeration tree.

When there are no more tuples in the TupleSpace and all threads are idle, Melinda sends a termination signal, and the program stops.

Depth-based cutoff : The parallelization strategy that we exploit for PGLCM is a classical strategy for tree-recursive algorithms. One of the issues of this strategy is that because of combinatorial explosion, it may create too many tasks, increasing the overheads of the underlying parallelism framework. Especially in our case, for low minimal support values, there will be many patterns, most of them being "deep" (in the enumration tree), corresponding to long, specialized patterns, supported by few transactions. For such patterns, computing *PGlcmLoop* is very fast due to the little number of transactions involved, this time gets close to Melinda's overhead time.

The solution is to coarsen the granularity level by adding a cutoff based on the depth in the enumeration tree, tracked by parameter d of PGlcmLoop : if this depth is upper or equal to a threshold value $maxD$ given in input, in line 29 of Algorithm 5, instead of putting tuples in the TupleSpace, a sequential call to *GlcmLoop* (Algorithm 2) is performed. This means that we consider that it is not necessary to create more parallel subtasks for the subtree of this pattern : the enumeration subtree is computed sequentially, making a bigger task where parallel frameword overheads are only paid once.

Locality issues : Another important issue for our parallelization strategy is locality. When a pattern Q is computed by *PGlcmLoop*, this pattern is stored in the cache of the processor, and all the elements of $\mathcal{T}$ and $\mathcal{B}$ that were used to compute Q's support are also in cache. These elements are likely to be necessary in order to compute the support of the patterns that are immediate children of Q in the enumeration tree. However after processing Q its children are pushed as tuples to the TupleSpace, and may be distributed to any idle thread, possibly breaking the locality principle.

In practice, Melinda uses a simple an efficient technique to remember which thread produced which tuple : the TupleSpace implementation is decomposed

into several deques, and each deque corresponds to a physical core. When a thread from this core makes a tuple request with *get*, it gets in priority tuples from the deque of the core. Only if this deque is empty, are deques of other cores examined for tuples. This method guarantees a best effort locality. More details can be found in the description of Melinda given in (Negrevergne, 2011).

4. Experiments

We present in this section an experimental study on the execution time and memory consumption of GLCM and PGLCM. We first present comparative experiments between our new algorithms and the current state of the art, Grite (sequential) (Di Jorio et al, 2009) and Grite-MT (parallel) (Laurent et al, 2010). The comparison is “unfair”: GLCM/PGLCM compute only the closed frequent gradual itemsets, whereas Grite/Grite-MT compute all the frequent gradual itemsets. However, there exist no algorithm (before GLCM) for mining closed frequent gradual itemsets. The experiments in the paper defining the notion of closure for gradual itemsets (Ayouni et al, 2010) rely on a post-processing of the results of Grite, which takes even more time than running Grite alone.

Thus, our experiments reflect the fact that up to now the only way to get gradual itemsets was to use Grite/Grite-MT, and we show the advantage of using our approach instead.

The comparative experiments are based on synthetic datasets produced with the same modified version of IBM Synthetic Data Generator for Association and Sequential Patterns as the one used in (Di Jorio et al, 2009; Laurent et al, 2010).

All the experiments are conduced on a 4-socket server with 4 Intel Xeon 7460 with 6 cores each, for a total of 24 cores. The server has 64 GB of RAM. We compare our C++ implementation of GLCM/PGLCM with the original C++ implementation of Grite/Grite-MT.

4.1. Comparative experiments: sequential

The first experiment compares the run time and memory usage for GLCM and Grite. The dataset used, C1000A20, has 1000 transactions and 20 items. Figure 7 shows the execution time for both programs when varying the support, with a logarithmic scale for time. The number of closed frequent gradual patterns returned by GLCM is compared with the number of frequent gradual patterns returned by Grite in Figure 8. The memory usage is shown in Figure 9.

The execution time results show that GLCM is two orders of magnitude faster that Grite for handling this small dataset: for the lowest value it answers in 29s, while Grite needs 1 hour and 40 minutes. This result directly comes from the fact that there are two order of magnitude less closed frequent gradual patterns than frequent gradual patterns. Both programs have a low memory usage on this small dataset. As expected GLCM memory usage is constant whatever the support value, while for lower support values Grite increases its memory usage, because it depends on the number of frequent gradual itemsets to find.

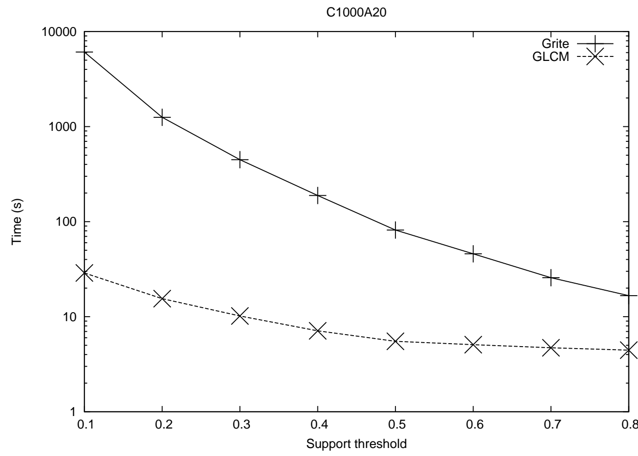


Fig. 7. Time vs support, sequential

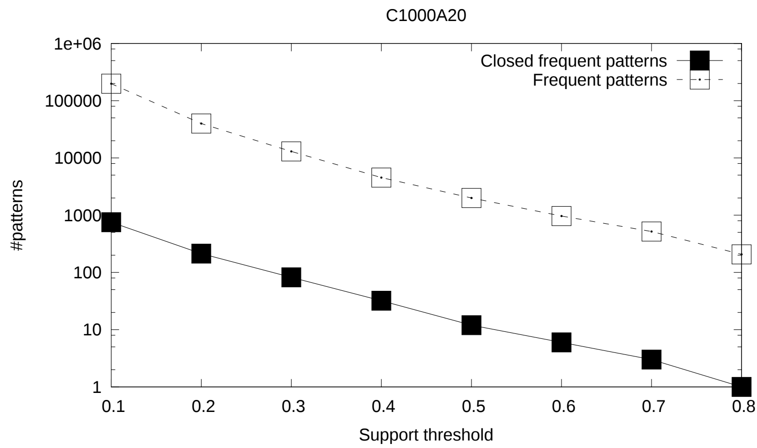


Fig. 8. Number of closed frequent gradual patterns vs number of frequent gradual patterns

4.2. Comparative experiments: parallel

The next experiment compares the scaling capacities of PGLCM and Grite-MT on several cores, for the dataset C500A50 with 500 transactions and 50 items. This dataset is more difficult than the previous one, as the complexity lies in the number of items which determines the number of (closed) frequent gradual itemsets.

For all parallel experiments, we exploit the depth cutoff method presented at

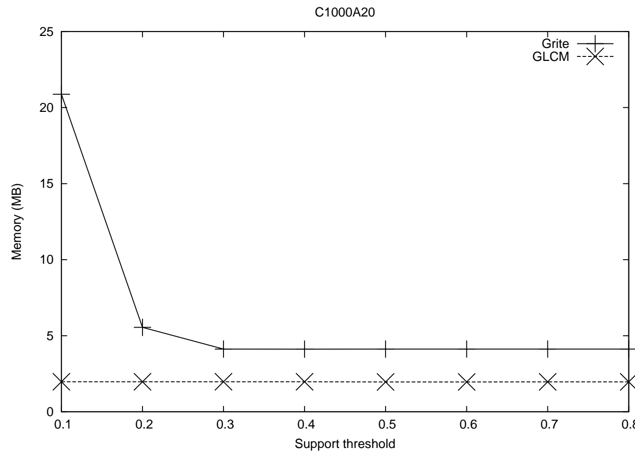


Fig. 9. Memory vs support, sequential

the end of Section 3.6. Setting $maxD = 4$ gave us the best overall results so this value is used for all experiments.

Figure 10 shows the execution time for both algorithms w.r.t. the number of threads, with a logarithmic scale for time. Figure 11 shows the speedup w.r.t. sequential execution for both algorithms. Last, Figure 12 shows the memory consumption. The suuport threshold is fixed at 0.9, for this value there are 114 closed frequent gradual patterns outputted by PGLCM while Grite-MT outputs 378447 frequent gradual patterns (3 orders of magnitude more).

In this experiment with a more complex dataset, PGLCM is again two orders of magnitude faster than Grite-MT. With all 24 threads, PGLCM completes execution near instantly in 1.4s, while Grite-MT needs 335s. The memory usage does not change much whatever the number of threads for both programs. Grite-MT exhibits a better speedup than PGLCM on this experiment. However, the run times for PGLCM get very low with more than 8 threads: they are between 1 and 2 seconds. There may not be enough work for PGLCM to exploit all 24 threads. More than a parallelism issue, this speedup difference comes from the difference of the problems handled by Grite-MT and PGLCM : by focusing on **closed** frequent gradual patterns PGLCM explores a far smaller search space than Grite-MT, which has a lot more work computing all frequent gradual patterns. This allows Grite-MT to give enough work to all the cores and have good speedup, but with a poor execution time compared to PGLCM.

We did the same experiment with C800A100, a more complex dataset with 800 transactions and 100 items. Grite-MT could not run for this dataset: it filled up the 64 GB of RAM of our machine and could not complete. This excessive memory consumption was already mentioned in (Laurent et al, 2010), and comes from the fact that memory complexity in Grite/Grite-MT, like in Apriori, depends on the number of frequent gradual itemsets to find. PGLCM does not have

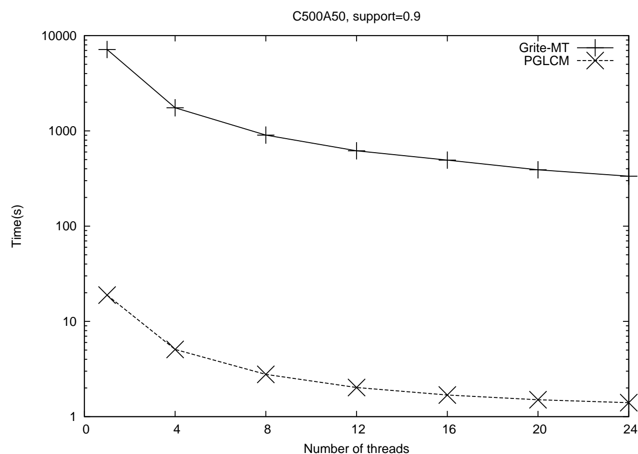


Fig. 10. Time vs #threads

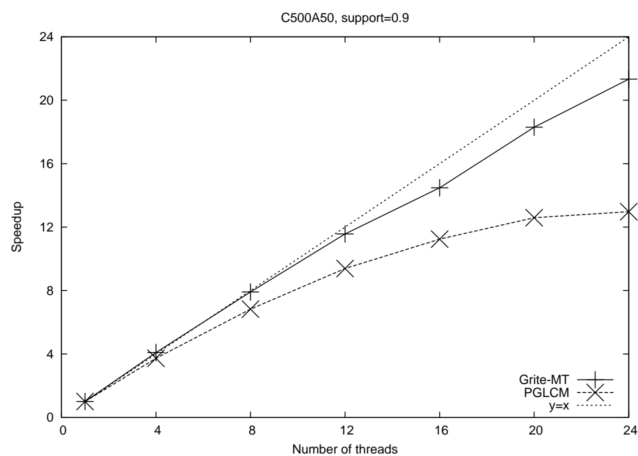


Fig. 11. Speedup

this problem, we thus report its run time in Figure 13, the speedup in Figure 14 and the memory consumption in Figure 15.

For this more complex problem the run time with 24 cores is 48s, so there is enough computation to keep the program busy. The speedups are far better in this case, with an excellent speedup of 22.15 for 24 threads. The granularity of our parallelization is well adapted to complex datasets. For further works, it could

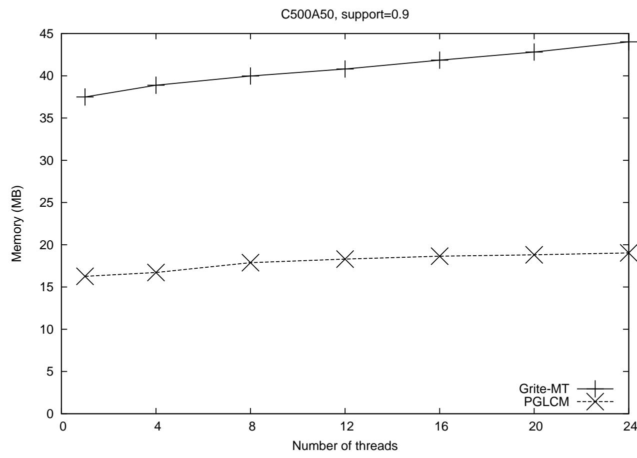


Fig. 12. Memory vs #threads

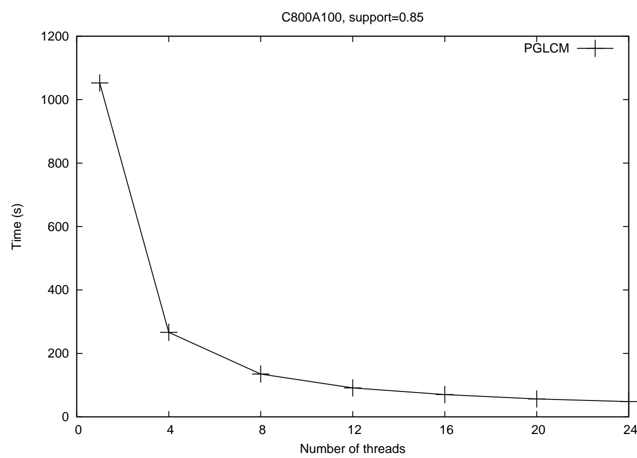


Fig. 13. Time vs #threads

be interesting to be able to decompose the computations in lower granularity tasks when faced with simpler datasets such as C500A50.

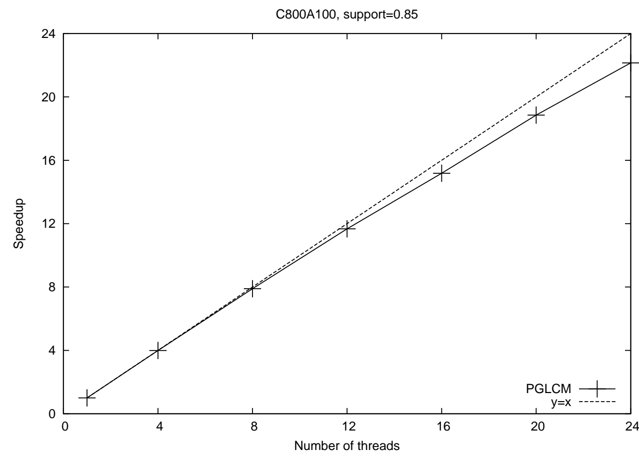


Fig. 14. Speedup

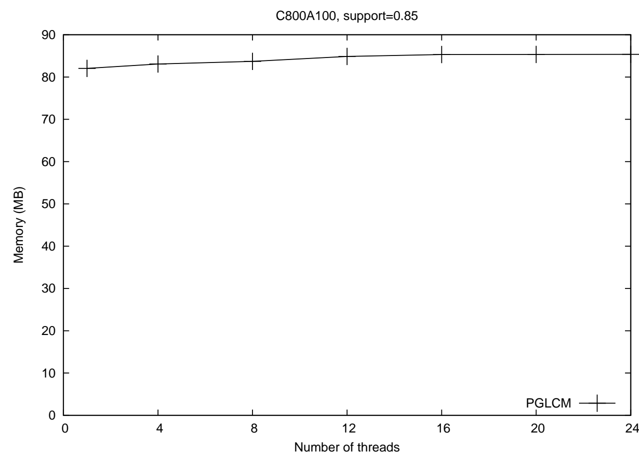


Fig. 15. Memory vs #threads

4.3. Qualitative experiment: mining user data in MovieLens

The major interest of our gradual pattern definition is that it is well adapted to discover some partial orderings on attributes on data in which no particular order pre-exists. One such kind of data that receives a lot of attention nowadays is *user data*, i.e. data produced by individual users, often in Web 2.0 applications. As

Pattern	Pattern description	Support
ML_1	$\{age^{\uparrow}, documentary^{\uparrow}\}$	1,459
ML_2	$\{age^{\uparrow}, documentary^{\uparrow}, western^{\uparrow}\}$	880
ML_3	$\{age^{\uparrow}, documentary^{\uparrow}, romance^{\downarrow}\}$	57
ML_4	$\{age^{\uparrow}, documentary^{\uparrow}, children's^{\downarrow}\}$	52
ML_5^{Male}	$\{age^{\uparrow}, mystery^{\downarrow}\}$	66
ML_6^{Male}	$\{age^{\uparrow}, sci-fi^{\downarrow}\}$	57
ML_7^{Female}	$\{age^{\uparrow}, sci-fi^{\uparrow}, mystery^{\uparrow}\}$	58

Table 8. Some gradual patterns from MovieLens

a typical example, consider the MovieLens[2] dataset, a real dataset consisting of ratings of movies by users. The dataset information contains for each user basic demographic information (age, gender) and for all movies rated by that user a numerical rating in $[1, 5]$. MovieLens also gives a genre for each movie, allowing to regroup movies in genres such as *Drama, Action,etc.* From the MovieLens 1M dataset (i.e. having 1 million ratings), we prepared an aggregated dataset where:

– each user is a transaction, there are 6,040 transactions
– there are 21 columns:
  · numerical user id
  · user age
  · total number of movies rated by user
  · average rating for user
  · average rating per genre for user (18 different genres)

We mined this dataset with PGLCM, for a minimal support threshold of 50. The computation completed in 6 hours and outputted 13,775 patterns.

Table8 shows some of the discovered patterns.

Such patterns show that orderings could be found in parts of data, indicating a trend that can help explain data's content. For example, gradual pattern ML_1 indicates that the older the user, the higher he/she will rank documentaries, this being valid on 24% of the users. This correlation between the age and the rating of documentaries is thus quite strong, and helps to better interpret documentary rankings. Gradual pattern ML_2 shows that for 14% of the users considered, together with documentaries' rating the rating of western movies is also higher for older people.

Due to the provided demographic data, it is even possible to separate the dataset into male users (4,331 users) and female users (1,709 users), mine each subdataset and search for a variant of discriminative patterns where opposite variations would be found in the two datasets (Cheng et al, 2007). Among the found patterns are ML_5^{Male}, ML_6^{Male} and ML_7^{female}: the male patterns show that for a part of the male users, an increase in age is correlated with a lower ranking for mystery or science fiction movies. On the other hand, both these categories get an increase in ranking with age for female users. One tentative explanation is that science fiction movies (and to a lesser extent mystery movies) lead to very polarized rankings for young people; young men love them leading to high ratings, while young women have no interest in them, leading to minimum

[2] http://movielens.umn.edu/

ratings. With age, this polarization tends to disappear, these movies becoming movies "like the others". This leads to reducing the average score for men and increasing it for women.

This experiment shows one of the advantages of our approach: in the dataset there was no a priori ordering of the users. The algorithm could take the dataset without any order on the rows and discover "hidden" orderings.

4.4. Qualitative experiment: mining stock market data

We have run PGLCM on US Stock Market data[3] in order to find out interesting knowledge and give potentially useful suggestions for people who intend to invest in a company. We gather all stock price for some companies' stock symbol and combine them in a large dataset. The lines of the dataset represent the date when the prices (in US Dollar) are retrieved. The first column stands for the date and all next columns of the dataset stand for each Stock Symbol (around 300 Stock Symbols).

We are only interested in patterns where the ordering of supporting transactions matches the temporal order. The date is thus given as a numerical attribute, and we only mine patterns of the form $\{Date \uparrow, *\}$, which guarantees the temporal ordering of the transactions supporting these patterns. This is done at the algorithm level by suppressing in Algorithm 2 the for loop of line 6, which is no longer necessary, and by calling *GlcmLoop* on line 7 with $e = 0$, which corresponds to $Date \uparrow$ when *Date* is the first attribute. This enforces the constraint while pruning a large part of the search space. For clarity, in the rest of this section $Date \uparrow$ is omitted from the patterns reported.

We focused our analysis on the time frame of the Deepwater Horizon oil spill[4] from 20 April to 15 July 2010 in a dataset of weekly quotes. We were especially interested by patterns comprising the quote of the BP company (NYSE: BP). We present the gradual patterns found in this dataset and our effort to explain them by searching in current events (newspapers, other sources). We saw that there are many companies affected by this accident together with BP. First, we found ($BP \downarrow$, $HAL \downarrow$, $RIG \downarrow$, $APC \downarrow$). It means that the stock price of BP decreases gradually together with the stock price of Halliburton (NYSE:HAL), Transocean (NYSE:RIG) and Anadarko (NYSE: APC) in the same time (42.86% in 14 total weeks). Halliburton was responsible for cementing the ocean floor while Transocean was the owner of the rig that BP was leasing. And Anadarko has 25 percent working interest in BP's Macondo Prospect[5]. The result is shown in Figure 16.

Anadarko was affected worse than Halliburton and Transocean with the pattern ($BP \downarrow$, $APC \downarrow$). Looking into our result (Figure 17), we can see that 64.29% (in total 14 weeks) of time BP's stock price and Anadarko's stock price decrease together.

But there are some companies that have the stock price increase while BP's decrease. For example, the pattern ($BP \downarrow$, $CLH \uparrow$) shows that Clean Harbor (NYSE:CLH)'s stock price increases in 42.86% (in 14 total weeks) of time together with BP's stock price (as shown in Figure 18). Clean Harbor is an environ-

[3] Source from Yahoo Finance! http://finance.yahoo.com/
[4] http://en.wikipedia.org/wiki/Deepwater_Horizon_oil_spill
[5] http://en.wikipedia.org/wiki/Anadarko_Petroleum_Corporation

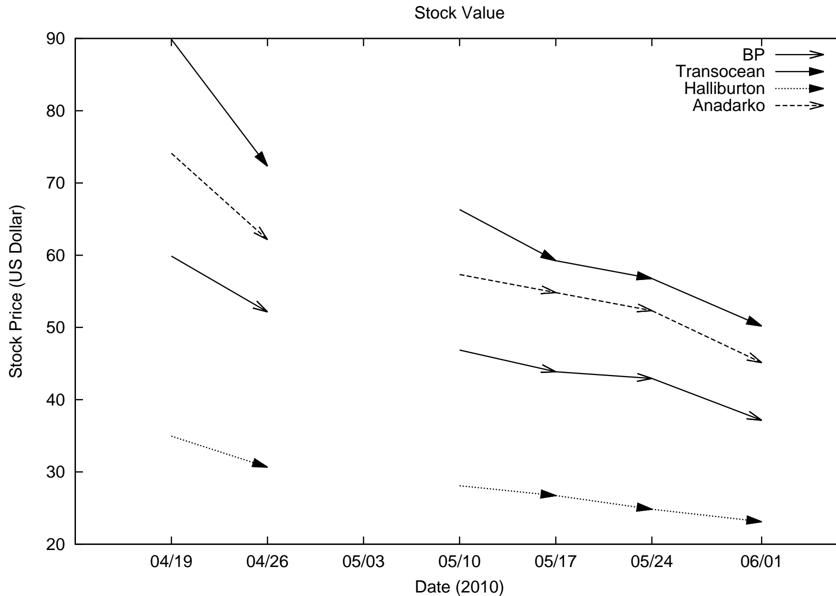


Fig. 16. BP, Halliburton, Transocean and Anadarko's stock price decrease in the same time

mental service provider focused in North America. Clean Harbor's crew members worked along the gulf coastline to mitigate the environmental damage of the BP oil spill. Another company's stock price that was expected to increase is Procter and Gamble (NYSE:PG) because it delivered 2,000 Dawn dish washing liquid bottles to the Gulf of Mexico in order to clean the affected wildlife in the region. But it had a small impact for a global company like Procter and Gamble and it did not affect P&G very much. We even found the pattern ($BP \downarrow$, $HAL \downarrow$, $RIG \downarrow$, $APC \downarrow$, $PG \downarrow$), presented in Figure 19), which shows that P&G action did not help it to rise its stock value.

5. Conclusion and perspectives

We have presented in this paper GLCM, the first algorithm for directly mining closed frequent gradual itemsets. Such gradual itemsets allow to find covarations between numerical attributes, with many applications to real data.

Our algorithm is based on the ppc-extension idea developed in the LCM algorithm, and which is currently the most efficient way to mine closed patterns, with a time complexity linear in the number of results to find and a memory complexity constant w.r.t. the number of results to find.

We also parallelized our algorithms in order to exploit the computing power of recent multicore processors.

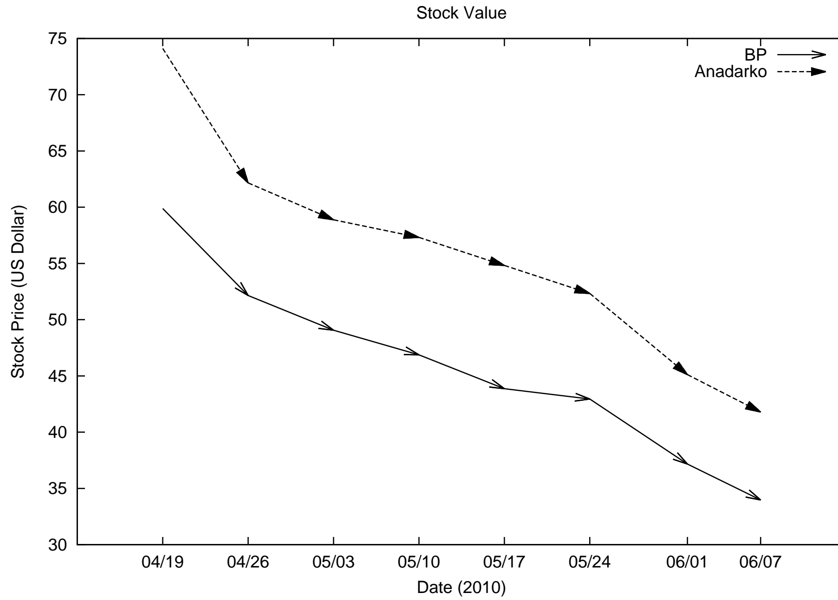


Fig. 17. BP and Anadarko's stock price decrease in the same time

Our experimental study have shown that our approach, either sequential or parallel, is two orders of magnitudes faster than the state of the art. Our parallel algorithm scales well with the number of available cores for complex datasets, where such computing power is really needed. The low memory requirements of our algorithms allow them to handle large real world datasets, which could not be handled by existing algorithms due to memory saturation. Our algorithms thus removed the lock that prevented the use of gradual patterns analysis in realistic applications. We have shown that it is able to find many interesting knowledge in real world datasets about movie ratings or stock market.

The code is available as Open Source at `http://membres-liglab.imag.fr/termier/PGLCM.html`.

Our work opens several perspectives. An immediate perspective is to cooperate with practitioners having large numerical datasets, in order to help them extracting and analyzing gradual datasets. Reporting the results of such experiments will allow to show the practical interest of gradual pattern and hopefully lead to further research in the field of gradual pattern mining.

Other perspectives lie in the improvement of our algorithms. Currently, our algorithm always uses the same binary matrices whatever the gradual itemset under consideration. However (Di Jorio et al, 2009) has shown that the binary matrices could be reduced: some transactions never appear in the support of a pattern, so the corresponding lines and columns can be suppressed from the

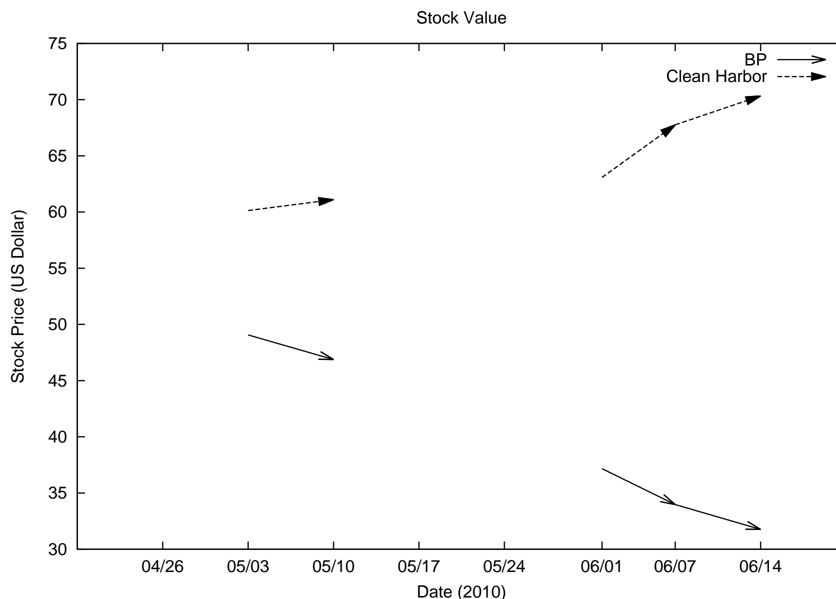


Fig. 18. BP's stock price decreases while Clean Harbor's stock price increases in the same time

corresponding binary matrix. This optimization would not reduce the theoretical complexity, however the FIMI workshop results (Goethals, 2003-2004; Uno et al, 2004) showed that reducing databases was one of the keys for reducing practical run time.

References

R. Agrawal and R. Srikant (1994), "Fast algorithms for mining association rules," in *Proceedings of the 20th VLDB Conference*, 1994, pp. 487–499.

J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu (2001), "Prefixspan: Mining sequential patterns by prefix-projected growth," in *ICDE*, 2001, pp. 215–224.

T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa (2002), "Efficient substructure discovery from large semi-structured data," in *In Proc. of the Second SIAM International Conference on Data Mining (SDM2002), Arlington, VA*, April 2002, pp. 158–174.

A. Inokuchi, T. Washio, and H. Motoda (2000), "An apriori-based algorithm for mining frequent substructures from graph data," in *PKDD*, 2000, pp. 13–23.

R. Srikant and R. Agrawal (1996), "Mining quantitative association rules in large relational tables," in *SIGMOD Conference*, 1996, pp. 1–12.

Y. Aumann and Y. Lindell (2003), "A statistical theory for quantitative association rules," *J. Intell. Inf. Syst.*, vol. 20, no. 3, pp. 255–283, 2003.

T. Washio, Y. Mitsunaga, and H. Motoda (2005), "Mining quantitative frequent itemsets using adaptive density-based subspace clustering," in *ICDM*, 2005, pp. 793–796.

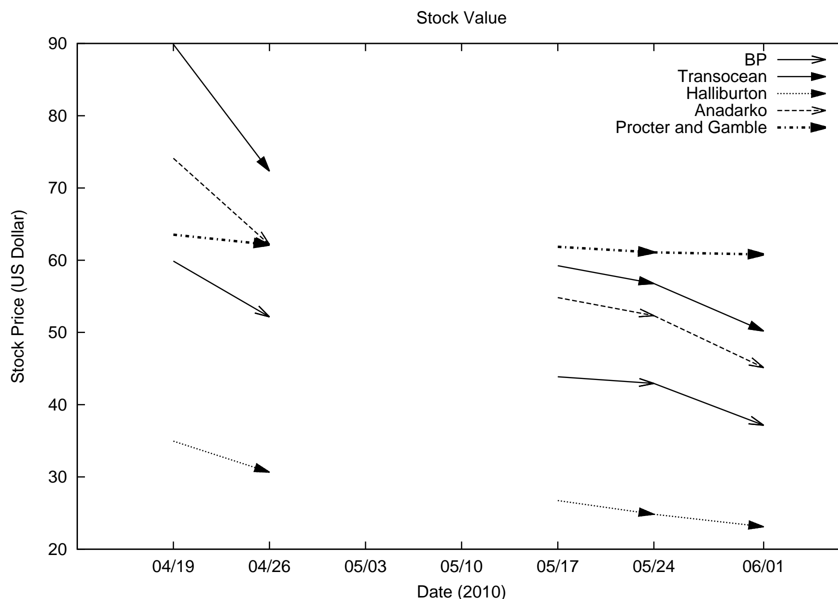


Fig. 19. BP and Procter and Gamble(P&G)'s stock price decrease in the same time

L. Di Jorio, A. Laurent, and M. Teisseire (2009), "Mining frequent gradual itemsets from large databases," in *Int. Conf. on Intelligent Data Analysis, IDA'09*, 2009.

N. Pasquier, Yves, Y. Bastide, R. Taouil, and L. Lakhal (1999), "Efficient mining of association rules using closed itemset lattices," *Information Systems*, vol. 24, pp. 25–46, 1999.

B. Goethals (2003-2004), "Fimi repository website," `http://fimi.cs.helsinki.fi/`, 2003-2004.

T. Uno, M. Kiyomi, and H. Arimura (2004), "Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets," in *FIMI*, 2004.

B. Negrevergne, A. Termier, J.-F. Mehaut, and T. Uno (2010), "Discovering closed frequent itemsets on multicore: Parallelizing computations and optimizing memory accesses," in *The 2010 International Conference on High Performance Computing & Simulation (HPCS 2010)*, 2010.

B. Negrevergne (2011), "A Generic and Parallel Pattern Mining Algorithm for Multi-Core Architectures," in *PhD dissertation*, 2011.

H. Arimura and T. Uno (2005), "An output-polynomial time algorithm for mining frequent closed attribute trees," in *15th International Conference on Inductive Logic Programming (ILP'05)*, 2005.

F. Berzal, J.-C. Cubero, D. Sanchez, M.-A. Vila, and J. M. Serrano (2007), "An alternative approach to discover gradual dependencies," *Int. Journal of Uncertainty, Fuzziness and Knowledge-Based Systems (IJUFKS)*, vol. 15, no. 5, pp. 559–570, 2007.

E. Hüllermeier (2002), "Association rules for expressing gradual dependencies," in *Proc. of the 6th European Conf. on Principles of Data Mining and Knowledge Discovery, PKDD'02*. Springer-Verlag, 2002, pp. 200–211.

A. Laurent, B. Négrevergne, N. Sicard, and A. Termier (2010), "Pgp-mc: Towards a multicore parallel approach for mining gradual patterns," in *DASFAA (1)*, 2010, pp. 78–84.

S. Ayouni, A. Laurent, S. B. Yahia, and P. Poncelet (2010), "Mining closed gradual patterns,"

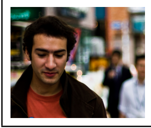
in *10th International Conference on Artificial Intelligence and Soft Computing, ICAISC 2010*, ser. LNCS, vol. 6113, 2010, pp. 267–274.
A. Laurent, M.-J. Lesot, and M. Rifqi (2009), “Graank: Exploiting rank correlations for extracting gradual dependencies,” in *Proc. of FQAS’09*, 2009.
M. Kendall and B. Babington Smith (1939), “The problem of m rankings,” *The annals of mathematical statistics*, vol. 10, no. 3, pp. 275–287, 1939.
C. Lucchese, S. Orlando, and R. Perego (2004), “Dci closed: A fast and memory efficient algorithm to mine frequent closed itemsets,” in *FIMI*, 2004.
T. Uno, T. Asai, Y. Uchida, and H. Arimura (2004), “An efficient algorithm for enumerating closed patterns in transaction databases,” in *Discovery Science*, 2004, pp. 16–31.
D. Gelernter (1989), “Multiple tuple spaces in linda,” 1989, pp. 20–27. [Online]. Available: http://dx.doi.org/10.1007/3-540-51285-3_30
D. Dubois and H. Prade (1996), “What are fuzzy rules and how to use them,” in *Fuzzy Sets and Systems*, vol. 84(2), pp 169–185, 1996.
D. Dubois and H. Prade, “Gradual inference rules in approximate reasoning,” in *Information Sciences*, vol. 61, pp. 103–122, 1992.
D. Dubois, H. Prade, and M. Grabisch, “Gradual rules and the approximation of control laws,” in *Theoretical aspects of fuzzy control*, pp. 147–181, 1995.
D. Dubois, H. Prade and L. Ughetto. “A new perspective on reasoning with fuzzy rules.” in *International Journal of Intelligent Systems*, vol. 18(5), pp 541–567, 2003.
H. Cheng, X. Yan, J. Han and C.-W. Hsu . “Discriminative Frequent Pattern Analysis for Effective Classification” in *International Conference on Data Engineering*, 2007, pp 717–725.

Author Biographies

Trong Dinh Thac Do received a B.E. degree in Computer Engineering from Ho Chi Minh University of Technology, Ho Chi Minh city, Vietnam in 2006 and a M.Sc. degree in Information Technology from Grenoble Institute of Technology (INPG) , Grenoble, France in 2011. GLCM/PGLCM is the Master Thesis work of Trong Dinh Thac Do, during his master internship between LIG laboratory, University of Grenoble, France and LIRMM laboratory, University of Montpellier 2, France. His research interests include research interests include data mining, database, parallel systems and information retrieval.

Alexandre Termier is Associate Professor at the University of Grenoble, France, at the LIG lab. As a member of the HADAS research group, he works on data mining, focusing on pattern mining. He is interested in parallelism in order to make scalable parallel pattern mining algorithms capable of exploiting multicore processors. He is also interested in mining structured data such as trees or graphs, especially in the context of the Semantic Web. Alexandre Termier teaches at the Computer Science Department of the University of Grenoble.

Anne Laurent is Full Professor at the University Montpellier 2, France, at the LIRMM lab. As a member of the TATOO research group, she works on data mining, sequential pattern mining, tree mining, both for trends and exceptions detections and is particularly interested in the study of the use of fuzzy logic to provide more valuable results, while remaining scalable. Anne Laurent teaches in the Computer Science Department at Polytech Montpellier Engineering School at the University Montpellier 2, which prepares a 5-year Masters in computer science and management.



Benjamin Negrevergne received his Ph.D. in 2011 from the University of Grenoble in France. He is interested in parallel and generic pattern mining algorithms. He is now working on constraint programming for data mining at the Dept. of Computer Science of KU Leuven in Belgium.

Behrooz Omidvar-Tehrani is currently a Ph.D. candidate in LIG laboratory (Grenoble, France) under the supervision of Sihem Amer-Yahia and Alexandre Termier. He is interested in interactive systems for data mining and pattern analysis methods. He obtained his master degree from the University of Grenoble in the field of Artificial Intelligence and the Web, and his B.E. degree from IAUM university, Iran.

Sihem Amer-Yahia is DR1 CNRS at LIG in Grenoble. Her interests are at the intersection of large-scale data management and analytics with an application to the social Web. Until July 2012, she was Principal Scientist at the Qatar Computing Research Institute where she led a group in Social Computing. Between May 2006 and May 2011, she was Senior Scientist at Yahoo! Research and worked on revisiting relevance models and top-k processing algorithms on Delicious, Yahoo! Personals and Flickr. Before that, she spent 7 years at at&t Labs in NJ, working on XML query optimization and XML full-text search. Sihem is a member of the VLDB Endowment and serves on the editorial boards of ACM TODS, the VLDB Journal and the Information Systems Journal. She is track chair of SIGIR 2013 and of PVLDB 2013. She will serve as PC chair of EDBT 2014. Sihem received her Ph.D. in Computer Science from Paris-Orsay and INRIA in 1999, and her Diplôme d'Ingénieur from INI, Algeria.

Correspondence and offprint requests to: Alexandre Termier, LIG, CNRS UMR 5217 Grenoble University, Grenoble, France. Email: Alexandre.Termier@imag.fr