Frequent sets, sequences, and taxonomies: new, efficient algorithmic proposals *

J. Baixeries
Data Mining Research Group
Synera Systems
jbaixeries@synerasystems.com

G. Casas
Departament de LSI
Universitat Politècnica de Catalunya
gcasas@lsi.upc.es

J.L. Balcázar
Departament de LSI
Universitat Politècnica de Catalunya
balqui@lsi.upc.es

December 19, 2000

Abstract

We describe efficient algorithmic proposals to approach three fundamental problems in data mining: association rules, episodes in sequences, and generalized association rules over hierarchical taxonomies. The association rule discovery problem aims at identifying frequent itemsets in a database and then forming conditional implication rules among them. For this association task, we will introduce a new algorithmic proposal to reduce substantially the number of processed transactions. The resulting algorithm, called Ready-and-Go, is used to discover frequent sets efficiently. Then, for the discovery of patterns in sequences of events in ordered collections of data, we propose to apply the appropriate variant of that algorithm, and additionally we introduce a new framework for the formalization of the concept of interesting episodes. Finally, we adapt our algorithm to the generalization of the frequent sets problem where data comes organized in taxonomic hierarchies, and here additionally we contribute with a new heuristic that, under certain natural conditions, improves the performance.

*This work is supported by ALCOM project (European Union ESPRIT LRT Project 20244)

1 Introduction

Data mining, or Knowledge Discovery in databases (KDD), is establishing itself as an emerging field that aims at finding useful information from large masses of data. One of the most revelant tasks in KDD is the discovery of association rules, as it was first formulated by [1], [3], and [2]. This task is often referred to as the *"market-basket"* problem since the analysis of supermarket sales or *basket* data is its most prototypical application. In this problem, we are given a (large) set of items and a (very large) set of transactions, each of which is a set of items. Each transaction may represent, in the prototypical example, the items purchased together by a customer in each individual customer transaction. The "association rule" discovery task identifies the groups of items that most often appear along with another group of items. For example, we may obtain the rule "60% of customers who buy bread and milk, also buy butter at the same time". There are many other applications such as predicting patterns in university enrollment, relating occurrences of words in text documents, and so on. As reviewed in more detail below, the algorithmically most interesting part of this problem essentially reduces the problem to finding those itemsets that are frequent enough.

There is a more general problem: finding "generalized association rules", where we take into account that, in many applications, items are naturally grouped together, e.g. as the various brands of the same or similar products; in most cases there is a large hierarchical taxonomy on the items that has to be accounted for. Frequent sets algorithms have to be adapted to this problem [14], [12], [10].

Another important task in KDD is the discovery of patterns in ordered data. In this problem, we are given a set of possible event types and a long sequence, which is made up of events. Every event in the sequence has associated a time occurrence, that is, the exact moment when that event appears. This imposes a total order on the occurrences of the events. The discovery of patterns in ordered data aims at analyzing sequences of events and discovering recurrent combinations of those events. These combinations of events are called episodes. Variants of frequent sets algorithms can be used for this purpose ([11], see also [4], [13]).

In this paper, the first problem we address is the performance issue of market-basket analysis. We improve performance over past methods by introducing a new proposal for finding frequent itemsets; it essentially consists of a nontrivial refinement of a state-of-the-art algorithm (DIC [6], see below). We build on the idea of reducing the number of necessary transactions that the algorithm needs to process in order to find the frequent itemsets in a database without affecting the computing time, and obtain two advantages with respect to our basis algorithm: an empirically faster algorithm, and one less parameter that the user has to tune to make the algorithm efficient.

To explore the flexibility of our proposal, we try and adapt it also to the case of mining episodes from sequences, and propose yet another alternative to select the successive episodes that we argue is better than existing ones. It is more intuitive than previous approaches and it certainly helps to improve the eficiency of the algorithms solving this task.

Finally, in our third problem we test again the flexibility of our approach by applying it to the generalized association rules problem, i.e. the case where there is a hierarchical taxonomy on the items of the database. This hierarchy classifies the items into wider sets of elements, which can also be grouped into wider sets, and so on. We propose yet another idea of similar flavor to the previous ones, obtain a new algorithm, and describe the results of some preliminary experiments that suggest that it is very advantageous to use it when the shape of the taxonomies has some properties regarding, essentially, the amount of branching present in the hierarchy.

2 Frequent Sets

In this problem we are given as data a set of transactions, where each transaction, in turn, is an unordered set of items. As indicated above, each transaction may represent, for instance, those items bought simultaneously by an individual costumer to a retail store.

The frequent sets task aims at finding those groups of items which are often purchased together by costumers: i.e., those sets of items that appear simultaneously in large fractions of the total of transactions. Then, from these frequent sets of items, we can form association rules.

In this section, we present a new proposal to find them. Then, we experimentally compare our new proposal with a previous approach, obtaining some significant improvements.

2.1 Preliminaries

Let $\mathcal{I} = \{i_1, i_2, \ldots, i_N\}$ denote a set of data *items*. A transaction, or also *itemset*, $\mathcal{T}$, is defined to be a subset of $\mathcal{I}$. A k-itemset is an itemset with k items. A database, $\mathcal{D}$, is a multiset of transactions, each of which has a unique identifier. The *support* of an itemset is the fraction of transactions in $\mathcal{D}$ that contain the itemset. An itemset is called *large* or *frequent* if its support exceeds a given threshold, σ.

Given m items, there are potentially 2^m frequent itemsets; however, only a small fraction of them are usually frequent. This empirical observation makes frequent sets algorithms useful.

An *association rule* is a conditional implication among itemsets, $A \rightarrow B$, where itemsets $A, B \subset \mathcal{I}$, and $A \cap B = \emptyset$. Each association rule receives a degree of "confidence" or "interest" based on ratios of the supports of A, B, and $A \cup B$, and there are several proposals about how to define them; then those rules having large enough such ratios are deemed potentially useful.

Thus the derivation of association rules can be broken into two main steps:

1. Find all frequent itemsets for a given threshold of support.
2. Form rules which exceed a given "confidence" or "interest" threshold from the frequent itemsets found in step 1.

In this paper we will mainly focus on the perfomance of the first step, which is by far the most computationally intensive step in the algorithm, and we do not discuss at all how to define confidence; see [6].

2.2 Previous work on finding frequent itemsets

Since the introduction of the problem of mining association rules ([1], [3], [2]) several algorithms have been proposed for the task of finding the frequent itemsets in a database. In this section we will just introduce those that are the main basis of our work.

The most computationally intensive step in the following algorithms is having to process the whole database several times to find all the frequent itemsets. However, whatever the algorithm used, it is always possible to apply sampling methods (as described e.g. in [15] and [8]) which can certainly reduce the number of passes througouht the data.

2.2.1 Apriori

Almost all the current algorithms for finding frequent itemsets use the property that all subsets of a frequent itemset must be frequent themselves. The most widespread algorithm using this property is **Apriori** [2]. This algorithm counts first the support of all the possible items (i.e., 1-itemsets), finding which ones exceed the user-specified threshold. Then it combines these frequent 1-itemsets to form the *candidates* of size 2, that is, the potentially frequent 2-itemsets, then it counts their support and determines which ones are the frequent 2-itemsets. It continues by combining those frequent 2-itemsets to form the candidates of size 3, counting their support and determining which are the frequent 3-itemsets; and so on.

Thus, only the frequent k-itemsets are used to construct *candidate* or *potentially frequent* $(k+1)$-itemsets. This process of combining all the frequent itemsets of the same size among them to form candidates of size one more is called *generation of candidates.*

For each size k, all the transactions of the database are processed; processing a transaction means incrementing the support count for all the current candidates that appear in the transaction. Thus, it is clearly seen that Apriori does as many passes over the database as the maximum number of elements in a candidate itemset, checking at pass k the support for all candidates of size k over the whole database.

2.2.2 DIC

The two main factors which control performance in Apriori are the number of passes over all the data and the efficiency of those passes. To address both of these issues the algorithm called **Dynamic Itemset Counting (DIC)** [6] was introduced. DIC performs better than Apriori because it reduces the number of passes made over the data. The intuition behind DIC is that it works like a train running over the data with stops at intervals M transactions apart (being

M a user-specified parameter). Therefore, every M processed transactions, the DIC algorithm "stops", and does a generation of all the possible candidates that can be already incoporated into the current pass over the database.

The main difference between Apriori and DIC is that in Apriori all itemsets of the same size, k, are handled in the same pass over the data. When arriving at the end of the database, they are combined to form the candidates of size $k+1$ for the next pass. In DIC, however, there is the flexibility of allowing itemsets of any size to enter in any stop as long as they fullfil the property of having all their subitemsets already frequent. Of course, each candidate has to get its support counted through the whole database. Hence, in DIC, every newly incorporated itemset must "get off the train" at the same stop the next time DIC goes around.

2.2.3 DIC versus Apriori

Broadly speaking, we can consider DIC a generalization (or: parameterization) of Apriori: when the interval M of DIC is set to the value of the length of the database, then DIC performs like Apriori. On the other hand, when the interval M is substantially less than the length of the database, then DIC is expected to perform better than Apriori. The reason is that DIC does fewer passes over the data than Apriori due to the advanced incorporation of candidates. Actually, if the data is fairly homogeneous throughout the file and the interval M is reasonably small, this algorithm is considerably faster than Apriori.

Indeed, the advanced incorporation of candidates most frequently makes DIC process fewer transactions and, in general, it can be stated that the smaller the value of M, the fewer processed transactions. In fact, a recursive analysis [5], [7] supports more formally that the maximum reduction in the number of processed transactions is when $M = 1$. Nevertheless, the frequent check for new candidates incurs in an overhead, and when M is set to a value close to 1, the generation of candidates must be done so often that, despite saving transactions, too much computing time is spent overall. Thus, both for large and for small values of M, Apriori works better.

We are left with the problem of determining, before running the algorithm, which is the proper value of M. Up to now, this value must be found in an experimental way; small M leads to less transaction processing but, unfortunately, this does not mean that the algorithm is faster then.

2.3 A new proposal: Ready and Go

The idea of this algorithm is to try to implement the "best of both" alternative of running DIC in the extreme case, that is, when the size of the interval M is 1, but, of course, without incurring in the overhead.

2.3.1 Ready and Go

The main idea of our algorithm, Ready and Go (R&G), is as follows. Every itemset will become candidate, and its support will start being counted throught the database, in the exact moment that it has all its subitemsets already frequent. To efficiently achieve this goal we should introduce a way to decide in constant time when an itemset has its subitemsets already frequent. The R&G algorithm maintains two collections of itemsets:

- The **candidate itemsets**, those itemsets whose support is being counted through the database.
- The **potential candidates**, those itemsets which are not being counted throught the database, yet some of their immediate subitemsets are candidate itemsets.

In the terminology and notation of [9], the set of potential candidates is exactly $Bd^{-}(S)$, that is, the negative border of S, where S is the set of all the itemsets whose support is being counted.

When a potential candidate has all its candidate subitemsets frequent, it will be immediately incorporated as a candidate so as to be counted through the database. To achieve this goal in constant time, both candidate itemsets and potential candidates will establish a kind of "communication". The idea behind this communication is that every potential candidate has associated a counter that indicates the number of its subitemsets which have been declared frequent. As soon as a candidate itemset of size k becomes frequent, it increments the counter of its suprasets of size $k+1$, thus notifying that it has achieved the state of frequent.

On the other hand, once a counter of one of these potential candidates indicates that every subitemset is frequent, it will become a candidate and will be incorporated into the pass over the database as a counting itemset. Since any set X of size k has k subsets of size $k-1$, when a counter of any potential candidate X of size k reaches the value k, then X will be incorporated as a candidate.

Each candidate itemset must get off of the counting after having visited all the transactions in the database. Simply, every candidate itemset must keep track of the transaction in which it was first incorporated, to get off there the next time R&G goes around.

Example 2.1 *In figure 1 we can see an example of how R&G works. In particular in this situation we have* $\mathcal{I} = \{A, B, C, D\}$ *as a set of data items. The set of counting itemsets is made up of* $\{A, B\}$,$\{A, C\}$, *and* $\{B, C\}$, *and every transaction marked in the database represents the point in which these counting itemsets become frequent.*

In the exact moment when the counting itemset $\{B, C\}$ *becomes frequent, it will notify to the supraset* $\{A, B, C\}$ *that it has achieved the state of frequent. At this stage,* $\{A, B, C\}$ *will have received the notification of all its subitemsets of size 2, so it will be time to enter into the counting: ready, and go.*

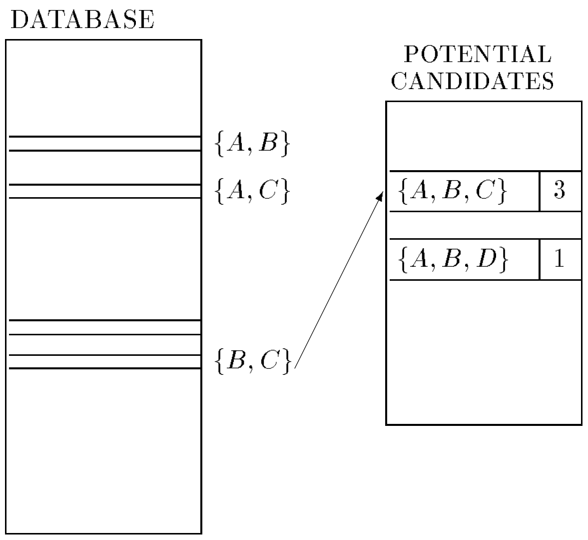

Figure 1: Example of how R&G works.

2.3.2 The data structures

The implementation of the R&G algorithm requires a data structure which can keep track of many itemsets and support some specific operations of itemsets. In particular, the data structure used for this, is exactly the hash tree used in, e.g., [6].

When it comes to the structure which can keep track of sets in the current negative border, that is, the potential candidates, the R&G algorithm just requires a hash table, so as to allow direct access for the notification of frequency. Every entry in this hash table represents one of those potential candidates which has a counter associated. The value of this counter is the number of subitemsets which have been already declared frequent. When a candidate itemset of size k is declared frequent, it notifies its state to every supraitemset of size $k+1$, which is either located in this hash table or added to it at this point. When the counter of a potential candidate of size j located in this hash table achieves the value of j, then this itemset will be immediatly incorporated into the set of candidates.

2.4 Results

We tested R&G on synthetic data generated by the IBM test data generator.[1] The synthetic data generator is well documented in [3] and it was very convenient to use. In our experiments R&G has been tested against DIC.

We used an average size of 10 items chosen from 100 items. According to

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

experiments reported in [6], it seems that a size of M around the 10% of the length of the database gives the best results for DIC, and this is the value used in all our experiments.

We focused our experiments in testing the behaviour of R&G when varying the following factors: *minimum support* and *length of the database*.

2.4.1 Varying the minimum support

For this experiment we used a 100.000-transaction synthetic database. We tried R&G and DIC for supports varying from 0.3% to 2%[2]. The results can be seen in table 1 and figure 2.

In particular, in table 1 we can see that R&G beat DIC by a range between 5% and 10% of the total number of visited transactions, in other words, R&G processes between 5% and 10% fewer transactions than DIC.

	DIC	**R&G**
2%	266.523	249.630
1.5%	266.523	253.695
1%	276.523	254.676
0.8%	276.523	264.007
0.6%	286.523	257.950
0.3%	294.784	277.867

Table 1: Total amount of processed transactions when varying the support.

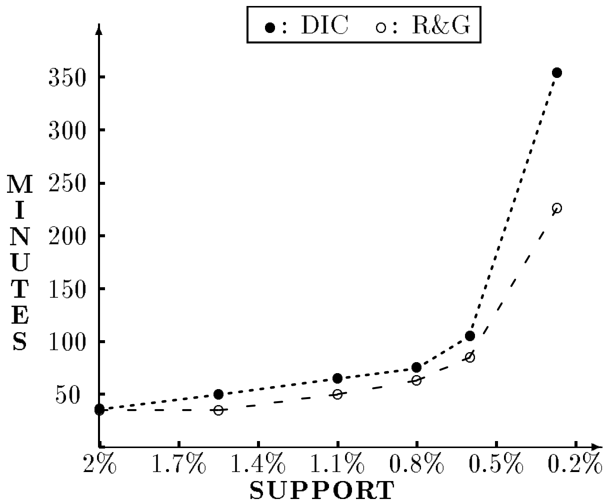

Figure 2: Time spent by the two algorithms given a minimum support.

The improvement, as far as computing time is concerced, is still bigger as

[2] With synthetic data, support threshold has to be considerably low.

we can see in figure 2. R&G beat DIC by about 20% of total time. Besides, the lower the support, the bigger the difference in total computing time.

2.4.2 Varying the length of the database

In this experiment, we wanted to see how the performance of R&G varies as the size of the database grows. Specifically, we were interested in checking whether the amount of spared transaction processing would grow proportionally along with the length of the database.

	DIC	**R&G**
100.000	286.523	257.950
250.000	691.369	648.332
500.000	1.382.807	1.299.898
750.000	2.074.291	1.941.000
1.000.000	2.765.721	2.581.855

Table 2: Total number of visited transactions when varying the length of the database.

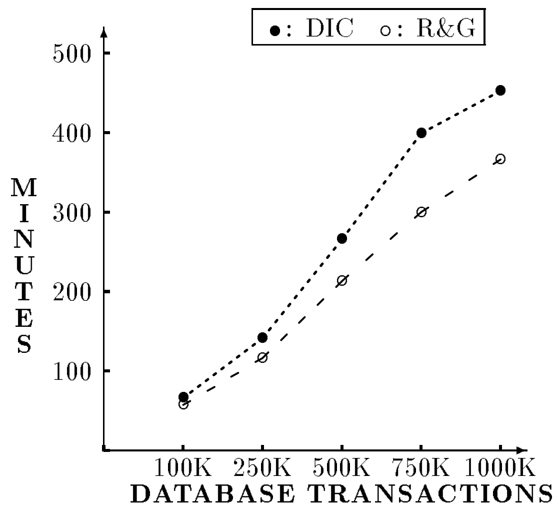

Figure 3: Total amount of time when varying the length of the database.

We used a minimum support of 0.6% in the whole experiment, and the sizes of the synthetic database were varying from 100.000 transactions to 1.000.000 transactions. The results can be seen in table 2 and in figure 3.

Table 2 shows the regularity of R&G performance. With respect to processed transactions, R&G processes around 10% fewer transactions than DIC. This was to be expected since the algorithm is designed to optimize this parameter. But still more important is the total computing time of the algorithm. These results can be seen in figure 3. Once again, R&G beat DIC by about 20% of the total computing time, thus showing that the timing overhead that DIC would incur by setting a low value of M does not appear in our algorithm.

Let us point out that only a small fraction of our large amount of experiments is explicitly reported here; the trends presented here were confirmed by the whole empirical process.

3 Mining Sequences

In this section, we try to move the ideas of the R&G proposal to a somewhat less developed area. In particular, the main goal of this section will be the discovery of patterns in case of having sequences of events instead of unordered collections of data. Examples of such data are alarms in a telecommunication network, crimes commited by a person, occurences of recurrent illnesses, etc. Abstractly, such ordered data can be viewed as a sequence of events, where each event has an associated time of occurrence. An example of an event sequence is represented in Figure 4. Here A, B and C are event types, that is, different types of alarms from a telecommunication network, or diferent types of user actions marked on a time line.

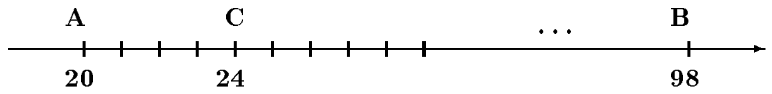

Figure 4: A sequence of events.

One basic problem in analyzing such sequences is to find frequent episodes, i.e, collections of events occurring frequently together in the event sequence. These frequent episodes are the interesting patterns in the sequence, and they are expected to provide useful information about it. Such information can then be used to better explain problems, to suppress redundant events or to predict severe faults.

In this section we describe an efficient solution to this problem of finding all these sequential patterns, based on the R&G algorithm. Besides this proposal, we also present a new approach for defining interesting episodes, which we consider more intuitive than previous ones.

3.1 Framework Formalization

First, we shortly review the definitions related with sequences. We follow the terminology, notation, and setting of [11].

3.1.1 Event sequences

The input of the problem is a *sequence* of events, where each event has an associated time of occurrence. Given a set E of event types, an *event* is a pair

(A, t) where $A \in E$ is an event type and t is its occurrence time.

An event sequence is a triple (s, T_s, T_e), where T_s is called the starting time of the sequence, T_e is the ending time, and s has the form:

$$s = \langle (A_1, t_1), (A_2, t_2), \ldots, (A_n, t_n) \rangle$$

where A_i is an event type, and t_i is the associated occurrence time, with $T_s \leq t_i < t_{i+1} \leq T_e$ for all $i = 1, \ldots, n-1$. For ease of intuition, we describe all facts as though each t_i is measured in seconds, but of course the time unit is actually irrelevant for our algorithms.

3.1.2 Episodes

Our desired output for each input sequence is a set of frequent episodes. An *episode* is a partially ordered collection of events occurring together in the given sequence. Episodes can be described as directed acyclic graphs. Consider, for instance episodes α, β and γ in Figure 5. Episode α is a *serial episode*: event type B occurs before event type C in the sequence. Of course, there can be other events occurring between these two in the sequence. The notation used for a serial episode will be: $B \rightarrow C$. Episode β is a *parallel episode*: events A and B occur frequently close in the sequence, but there are no constraints about the order of their appearences. The notation used for a parallel episode will be: $\{A, B\}$. Finally, episode γ is an example of *hybrid episode*: it occurs in a sequence if there are occurences of A and B and these precede an occurrence of C.

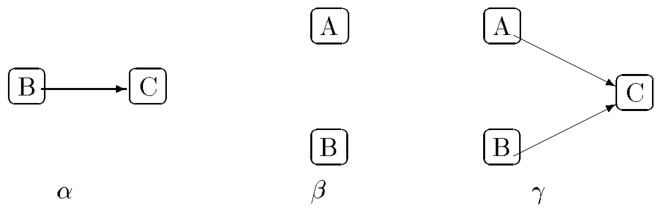

Figure 5: Types of episodes

More formally, an episode can be defined as a triple $(V, \leq, g)$ where:

- V is a set of nodes.
- $\leq$ is a partial order relation on V.
- $g : V \rightarrow E$ is a mapping associating each node with an event type.

The interpretation of an episode is that events in $g(V)$ must occur in the order described by $\leq$. An episode α is parallel if the partial order $\leq$ is trivial (i.e., $x \not\leq y$ for all $x, y \in V$ such that $x \neq y$) and an episode α is serial if the relation $\leq$ is a total order (i.e., $x \leq y$ or $y \leq x$ for all $x, y \in V$).

We also define the size of an episode as the number of events it contains, i.e, $|V|$. In this paper we will mainly deal with serial and parallel episodes.

3.1.3 Windows in a sequence

In the analysis of sequences we are interested in finding all *frequent* episodes from a class of episodes which can be *interesting* for the user.

To be considered interesting, the events of an episode must occur close enough in time. In [11] the user defines how close is "close enough" by giving the width of the *time window* within which the episode must occur. To be frequent, an episode must appear an specific number of times in the sequence, i.e, its number of appearences must be over a minimum user-specified value; in the approach of [11], this depends again on the same window width.

Generally speaking, and following the mentioned approach, an episode is frequent according to the number of windows where that episode has occured over the total possible windows in the sequence. However, this and other approaches given by [11], which we will describe later, present some disadvantatges we will try to avoid. For this reason, in this paper we will propose a diferent mesure of interestingness for an episode.

3.2 Previous work on sequences

This section introduces some of the most important algorithms and approaches used to mine sequences as they were described in the above sections. It is important to highlight that, apart from the presented problem formulation, there exists other approaches for sequences. The main one is the introduced by [4] and also studied by [13]. In this approach, the database is a collection of *customers*, each with a sequence of transactions, each of which is an itemset. In this case, the frequency of an episode α is the number of customers in the database whose sequences contain α as a subsequence. However, we will only focus on the problem definition first mentioned.

Thus, mainly, in this section we will take the work of [11] as a reference. In fact, in this work the most fundamental algorithm is Apriori and the two following aproaches are proposed.

3.2.1 WINEPI

In this first approach an episode is considered interesting if it fits into a specific window width. On the other hand, an episode is considered frequent according to the number of windows in which it occurs.

In WINEPI the user defines a fixed window width. The number of possible windows of width win that can be formed in the sequence (s, T_s, T_e) is exactly: $T_e - T_s + win - 1$. Thereby, the frequency of an episode α in s is defined to be:

$$fr(\alpha, s, win) = \frac{windows\ in\ which\ \alpha\ occurs}{total\ windows\ of\ width\ win\ from\ s}$$

The approach consists in applying the Apriori algorithm in sequences as though a window were a transaction with ordered events. The goal is finding out which are the interesting frequent episodes; so, in every processed window, the frequency of those episodes that fit in the window is incremented whereas the frequency of those episodes that do not occur in the window is decremented. The algorithm starts by computing the frequency of episodes of size 2, which are candidates to be frequent. After this stage, the frequent episodes from this set of candidates, are combined to form candidates of size 3, and so on. An episode α will be frequent when $fr(\alpha, s, win) \geq minfr$.

3.2.2 MINEPI

MINEPI is based on *minimal occurrences* of episodes in a sequence. For each frequent episode, the algorithm finds the location of its minimal occurrences.

The minimal occurrences with their time intervals are identified in the following way. Given an episode α and an event sequence s, we say that the window $w = [t_s, t_e)$ is a *minimal occurrence* of α in s, if:

(1) α occurs in the window w.

(2) α does not occur in any proper subwindow on w.

The maximum width of a minimal occurrence is given by the user. This width will be fixed during the process, and it will measure the interestingness of the episodes. Basically, the applied algorithm is Apriori: it locates, for every episode going from the smaller ones to larger ones, its minimal occurrences. In the candidate generation phase, the location of minimal occurrences of a candidate episode α is computed as a temporal join of the minimal occurrences of two subepisodes of α.

This approach differs from WINEPI in the fact that it does not use a frequency ratio to decide when an episode is frequent. Instead, an episode will be considered frequent when its number of minimal occurrences is over an integer value given by the user. This is a consequence of the fact that the lengths of the minimal occurrences vary, so that a uniform ratio would be misleading.

3.3 A new approach for sequences

The above described previous work on sequences has some disadvantages we wanted to avoid in our algorithms to mine sequences. This fact led to the study of a new approach for interesting episodes that will be described in this section.

3.3.1 Some disadvantages of the previous approaches

We summarize below some of the observed disadvantages in WINEPI and MINEPI.

- In both WINEPI and MINEPI the window width is fixed by the user and it remains fixed throughtout the mining. Consequently, the size of the discovered episodes is limited. For instance, if the window width is fixed with a value of 3 seconds, the maximum size of the discovered episodes is of 3 events, which are exactly the number of events that can fit in the window. Additionally, this requires the end user to fix one more parameter with not much guidance on how to do it.
- Both WINEPI and MINEPI can give rise to overlapping episodes. This fact is due to the window width, which is fixed and limited during the processing. When the user defines a width, he is fixing, at the same time, the size of the discovered episodes. So, if there exists an interesting episode, α, whose size is larger than the fixed window width, then that episode will never fit in any window and, consequently, α will be discovered just partially. For instance, if the window is of width 3 seconds and a larger frequent episode in the sequence is $A \to B \to C \to D$, the result of the mining will be: $A \to B \to C$ on the one hand, and $B \to C \to D$ on the other hand. These two episodes are overlapped parts of the same episode.
- Both previous approaches are using just the Apriori algorithm. (This is not difficult to change.)
- MINEPI does not use a frequency ratio to decide whether an episode is interesting. This makes difficult the application of sampling.

3.3.2 Proposed approach for interesting episodes

In order to avoid all these drawbacks and be able to enlarge the window width automatically throughout the mining process, we propose the following approach. We will consider a serial or parallel episode interesting if it fulfills the following two properties:

1. Its *correlative events* are separated at most *sec* seconds between them:

$$\underset{\alpha}{A \xrightarrow{sec} B \xrightarrow{sec} C} \qquad\qquad \underset{\beta}{\{A, \overset{sec}{\;} B, \overset{sec}{\;} C\}}$$

2. It is frequent.

So, in our proposal, the measure of interestingnes is *sec*, the number of seconds between correlative events. This number of seconds must be specified by the user. The above two episodes α and β are examples of the interpretation of our approach. In the serial episode α, the distance in seconds between A and B is *sec*, and the distance between B and C is also *sec* seconds. Besides, despite not specifying the distance between events A and C, it can be clearly seen that between A and C there is at most $2 \times sec$ seconds. In the parallel episode β, distance between correlative events A, B and C, regardless of the order of their appearences in the sequence, must be of at most *sec* seconds.

This new proposal makes the discovery of episodes a bit different. In previous approaches the user specified a fixed window in which episodes should be searched. The algorithm processed all the windows in the sequence and those episodes that did not fit in the window were not counted as an occurrence. We have now to distiguish episodes in a sequence, without restricting ourselves to such prefixed window widths.

Let us consider some specific subepisodes, that we call "parts", of a serial episode. These are subsequences of adjacents events. For example, the serial episode $A \to B \to C$ will have two parts of size 2, that is: $A \to B$ and $B \to C$.

Thus, we can formalize our problem in the following way: we will discover frequent episodes whose parts of size 2 can fit in a window of size *sec* seconds, or, equivalently, whose correlative events are separated at most *sec* seconds between them.

Since a part of size 2 of any episode contains 2 correlative events of that episode, and correlative events in a episode can have a separation of at most *sec* seconds, then, every part of size 2 of an episode will be searched in a window width of *sec* seconds. Hence, an episode with k parts of size 2 will be searched in windows of width $k \times sec$ seconds in the sequence.

Generally speaking, every episode that is candidate to be frequent, will be searched in windows whose width will be delimited by the episode size; equivalently, by the number of its parts of size 2. Thus, an episode with e events will be searched in all windows in the sequence of width $(e - 1) \times sec$ seconds.

With this approach the frequency of an episode can be defined in the following way. Let us denote by $W_k(s, win) = T_i - T_f + win - 1$ the total number of windows in a sequence (s, T_i, T_f) of a fixed width $win = k \times sec$ seconds. Then:

Definition 3.1 *The* frequency *of an episode α of size $k + 1$ in a sequence (s, T_i, T_f) is:*

$$fr(\alpha, s, win) = \frac{|\{w \in W_k(s, win) | \alpha \ occurs \ in \ w\}|}{W_k(s, win)}$$

To sum up, every episode α will be frequent if its frequency is over a minimum user-specified frequency, that is, according to the number of windows in which it occurs. However, in our approach the width of that window depends on the number of events in α, or what is the same, the number of parts of size 2. So, the effect in the algorithm is that while an episode size becomes bigger and the number of its events increases, the proper window in which that episode is searched, also increases its width; and simultaneously the ratio that has to be compared with the user-specified desired frequency is appropriately adjusted.

3.3.3 Advantages of our approach

Here we shortly summarize some advantages of our proposal for episodes.

- Since the window increases its width along with the episode size, the final frequent episodes do not overlap and their size is not limited.

- The association rules that can be mined from a sequence can contain more information because the discovered episodes that form that rule can be larger.
- All episodes found by WINEPI with a window width of x seconds can be found with our approach using a distance of $x - 1$ seconds between correlative events.
- We can define the frequency of an episode, which eases the application of sampling.
- Our proposal can be adapted to other types of sequential data, such as those in the approach of [4] and [13].

3.4 Seq-Ready&Go

In this case in which the algorithm must look for episodes, the basic ideas of R&G can be easily applied. So, the main aim of this section is to propose an improvement of the algorithms to mine sequences, which were based on Apriori. Of course, we will also incorporate our approach for defining frequent interesting episodes in the new algorithm.

To follow the same structure, this new algorithm, called Seq-Ready&Go (SeqR&G), will distiguish episodes in two possible ways:

- The **candidate episodes**, those episodes which form the set of current candidates and whose frequency is being counted through the sequence.
- The **potential candidates**, those episodes which have some subepisodes which are candidates.

The SeqR&G algorithm, like R&G, will incorporate the future candidates to the counting as soon as they are ready. In case of mining unordered data, a future candidate itemset was ready to be counted when all its subsets were frequent. Hovewer, in case of sequential data, and given that we are using our approach to interesting episodes, we must redefine once again the concept of "being ready".

In this algorithm, an episode α of size k *is ready* to become a candidate episode and so, begin the counting of its frequency in the sequence, when α has all its parts of size $k-1$ frequent instead of all its subepisodes. The parts of size $k-1$ of an episode α are a subset of all the subepisodes of size $k-1$ of the same episode α; that subset depends on whether α is serial, parallel, or hybrid.

So, we will deal with the parts of an episode instead of all its subepisodes. This is due to our mesure of interestingness, that is, fixing *sec*, the maximum time between correlative events. Because of this measure, any episode α must take into account only those subepisodes of α whose events were located in a correlative way in α. Those subepisodes of α that are located in a correlative way in α are the parts of size $k-1$ of α.

3.4.1 Discovering serial episodes

For serial episodes, the algorithm SeqR&G goes in the following way. It starts by initializing the set of candidate episodes with all episodes of size 2, and the set of potential candidates with all episodes of size 3. Then, it goes on counting the frequency of all the candidate episodes until this set becomes empty. When one of these candidate episodes of size k achieves the state of frequent, it increments the counter of all the potential candidates of size $k+1$ of which it is a part. On the other hand, when a potential candidate of size k has all its parts of size $k-1$ declared frequent it will be incorporated in the set of candidate episodes. Upon reaching the end of the sequence, the process starts over from its beginning like in the case of DIC. We know that a candidate is not frequent when the wrapped-up search comes to the point where counting started for that candidate.

It is important to highlight that, in this algorithm, the set of candidate episodes can be made up of episodes of diferent sizes, and SeqR&G must also be able to process windows of different sizes for every episode. Nevertheless, this problem is not difficult to solve: since an episode α of size k must be searched and counted in all windows of width $(k-1) \times sec$ seconds, SeqR&G will take, at every step, the largest window for the largest episode in the set of candidate episodes. The rest of episodes in the set of candidate episodes will be searched in the proper subwindows.

3.4.2 Discovering serial and parallel episodes simultaneously

In case of mining parallel episodes the problem can be reduced efficiently to mining serial episodes in the following way. Every parallel episode of size k lumps together $k!$ serial episodes. For instance, the parallel episode $\{A, B\}$ gathers the following two serial episodes: $A \rightarrow B$ and $B \rightarrow A$. In this case, a serial episode will be called *participant* of a parallel episode. Clearly, any serial episode is participant of one, and only one, parallel episode.

SeqR&G will mine serial episodes, but these serial episodes can refer to parallel ones too. Thereby, the set of candidate episodes will be made of serial episodes, while the set of potential candidates will be composed of parallel episodes.

According to our proposal, in order to be frequent, a parallel episode α must fullfil one of the two following conditions:

1. By adding up the frequency of the serial episodes that are participants of α, we reach the user-specified minimal frequency, and we say that α becomes frequent, *but* no serial episode participant of α is frequent alone; or

2. Every serial episode participant of α is frequent.

The reasons to propose these two new conditions are the following. Condition 1 reflects the case in which no serialized form of α is frequent, but the aggregation of all those non-frequent episodes into a parallel one, makes a contribution of useful information. Note that if one of them is already frequent, then α is giving less information, and we should not move from the serial episode to the parallel one. Condition 2 is different: here the parallel episode serves as a gathering of all the different forms in which the serial episodes are actually frequent.

Moving now into the algorithm, the parallel episodes, which are potential candidates, will simply wait for the notification of all their serial participants, which are the candidate episodes. This means that the algorithm will be counting the support of serial episodes, as in the previous case; however, when declaring one of these serial episodes, α, frequent or non-frequent, the notification must go to that parallel episode which α is participant of.

So, once one of these potential candidates, which are parallel episodes, has received the necessary notifications, it will be incorporated into the counting in the form of all their serial episodes participants.

4 Finding frequent itemsets with taxonomies over the database

In this section, we describe how to apply the R&G algorithm to the task of finding frequent sets in case of having a hierarchical taxonomy on the items of the database; beyond the simple direct applications, we will also propose a heuristic to improve the efficiency of the solution.

The main difference with the frequent sets problem is that we allow all elements in the taxonomy to appear in the frequent sets. For instance, if we had a taxonomy as in the example of figure 6, we could have frequent itemsets like {canned meat, apples}. It is important to realize that if the former set is frequent, the set {meat,apples} is also frequent, but this fact is redundant. We are interested only in the itemsets that contain the maximal information, so we want our algorithm to forget about the redundant itemsets. This results in an increased complexity of the process.

There are many applications, in particular the typical Market Basket Data Analysis, where such hierarchies have to be taken into account. Products are usually identified in transactions through their bar codes, which are different for different brands. Thus, the fact that some generic product like canned meat appears frequently together with something else may be difficult to see, because possibly each particular brand of canned meat is not frequent enough.

The algorithms to find frequent itemsets when there is a taxonomy over the database tend to reduce the problem to the basic frequent itemsets problem; this is done by enlarging each record with elements of the hierarchy (see below). More detailed information can be found in [12]. Also, specific algorithms have been developed [10]. The main idea behind these algorithms is that information contained in the hierarchy can be used to determine if a set can be frequent or not.

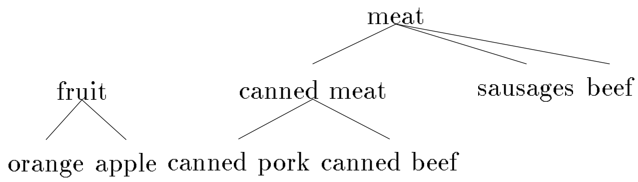

Figure 6: Example of taxonomy

4.1 Notation

First we review the notation used for databases with hierarchies. Let $I = \{a_1, a_2, \ldots, a_n\}$ be a set of items that can appear in our database DB, and let $T = \{t_1, t_2, \ldots, t_m\}$ be a *taxonomy*. This taxonomy represents the way that elements of I are grouped into more general sets: the elements of T are simply used as labels of these general groups. Now, let J be a hierarchy, which can be represented as an acyclic directed graph. The vertices of J are $I \cup T$. An edge in J from x to y means that x is the *parent* of y. (Also, we say that y is a *child* of x.) If there is a path from x to y, we say that x is an *ancestor* of y, and we use the notation $\hat{y}$, to denote any of the ancestors of y. There are a couple of natural restrictions:

- A vertex from I cannot be a parent of any other vertex in J.
- A vertex of J that does not belong to I must be the parent of at least one vertex.

Is is worth to note that:

- The support of an ancestor is not the sum of the supports of all its children.
- If $\{x, y\}$ achieves the minimum support, the following combinations also reach the minimum support:

$$\{\hat{x}, y\}, \{x, \hat{y}\}, \{\hat{x}, \hat{y}\}$$

- If the rule $x \to y$ has the minimal confidence, so does the rule $x \to \hat{y}$, but the rules $\hat{x} \to y$, and $\hat{x} \to \hat{y}$ may not.
- Rules $x \to \hat{x}$ are trivial, and provide no information at all.

4.2 Current approaches

To find frequent itemsets in a database with hierarchies over the data, the natural first approach is as follows: we could create a new database extending each of the transactions of the former database with all the ancestors of all the items appearing in the transaction; then we can apply any known frequent sets algorithm to mine this new database. We only should pay attention to avoid duplicates at each step of the algorithm, and to keep the most specific sets (we describe this in more detail below).

In fact, this simple framework is used in several algorithms, like in the algorithms *Basic* [12] and *Prutax* [10]. This framework has two good properties:

- It is conceptually easy to undestand and it is intuitive.
- Any algorithm can be applied to the enlarged database. Apriori is the usual choice, but other algorithms have been successfully adapted.

The enlarged transactions can be programmed to be made "on demand", so that the enlarged database never actually exists in the system. This allows one to avoid the increase of size of the database and to control the associated increase in processing time. We have followed this approach with R&G. However, this approach also has drawbacks:

- The hash tree of the candidate sets is increased.
- Care must be taken to avoid that redundant sets are counted at the same step of the algorithm.

We are now listing the main ideas that are implemented in some of the most well known algorithms used to mine databases with hierarchies, so that later we can propose a new alternative.

- Enlarging transactions, maybe virtually "on demand". This is found in all the algorithms.
- Filtering ancestors when the enlarged record is generated. Sometimes it is not necessary to include all the ancestors in the same record. For instance, if in our hierarchy A'' is the parent of A', and A' is the parent of A, if we are counting the set $\{A'', B\}$, for each record in which A appears, the enlarged record will have to include A'', but not A nor A'. This is implemented in *Cumulate* and *Prutax*.
- Itemsets in which x and $\hat{x}$ appear, are not generated. It is based in the following properties:
  - The support of $\{x_1, x_2, \ldots, x_n\}$ is smaller than or equal to that of $\{\hat{x_1}, x_2, \ldots, x_n\}$.
  - If in a set of size n, there are no pair of elements (x and y) such that x is an ancestor of y, when this set is later used combined with another set to form a set of size $n+1$, this new set will not contain any pair of elements (x' and y') such that x is an ancestor of y'. We thus can avoid the creation of sets in which there are ancestors of some of the elements in the same set, avoiding this possibility when creating the first sets.

This is implemented in *Cumulate* and *Prutax*.

- Pruning of the elements whose parent is not frequent. This is found in *Prutax*.

- A more sophisticated approach is found in *EstMerge*. In this algorithm the support of the elements located at the top of the hierarchy is first counted, then the support of the elements located one level beneath and whose parents are frequent are counted, and so on, until reaching the bottom of the hierarchy. In the case of EstMerge, the counting of the supports is made over a sample of the data in the database. To avoid the possible errors due to the sampling, the elements that are first considered to be frequent are those that not only are frequent, but also those that are not frequent but whose parents are all frequent. The last step of this algorithm consists in counting the frequent itemsets of $I \cup T$.

In general, the two drawbacks of all the existing algorithms that follow this approach are the important increase of the size of the internal data structures, which affects the speed of the algorithm very much, and the fact that the use of the hierarchy is somewhat of a gamble: by counting the support of an internal node of the hierarchy, with luck we can dispose of the counting of all the children, but with bad luck it is just wasted work.

4.3 Our alternative

Now, our new proposal can be summarized in the following points:

- We first calculate *all* the frequent itemsets (not only those of size 1!) formed by *only* internal nodes of the hierarchy, that is, all the elements of I.

- We then calculate the frequent itemsets of all the elements in the database, both leaves or internal nodes of the taxonomy, but with one restriction: when a new candidate itemset is created (let it be X), we check whether a more general itemset has been found frequent in the previous step. By "more general" of course we mean an itemset Y such that each element of X either belongs to or has an ancestor in Y.

- We accept to trade (on demand by the user) precision for speed; this is done by using a *different* frequency threshold for the two phases.

The first point is not completely trivial but, in some sense, is a natural thing to do. As indicated above, some existing algorithms do exactly this but only for items, that is, restricted to itemsets of size 1; we simply remove this restriction. The second point is its natural consequence. The third point is a tricky consequence of the other two: we could think about what happens using different support thresholds. We explain this point next.

4.3.1 Why increased support

Usually, the support of an ancestor is greater than that of any of its children. This implies that items whose parents have a support near the threshold are actually unlikely to reach the threshold. For instance, let's suppose that the minimal support requested is 5, and that we have the set $\{x\}$ (where x is an internal element of the hierarchy) whose support is exactly 5. We immediately see that if x has more than one child (which will be the usual case) then none of its children will reach the minimal support.

On the other hand, increasing the threshold on the support in the first step can produce errors, since there is the possibility that sets that do not reach this increased support are actually frequent (with respect to the original support) and do have some frequent children. Thus, the use of an increased support may result in the loss of some frequent itemsets; but only those that in turn are slightly above the support threshold. There may be cases where this degree of precision is not necessary for the actual task at hand, and where the user is willing to pay this price if the slightly incomplete results come out sufficiently earlier.

Thus we are left with the question of whether this approach is actually faster, since it has also the "gambling" component described above for simpler cases. In addition, it is unclear how much we can increase the frequency threshold without making useless the whole task, and how much we gain from each small additional increase.

Note that the second step can only be faster, and not slower, due to the presence of the first step, since there are less itemsets that are added as candidates to be counted. Thus, with respect to the efficiency issues, the challenge is to make the first step fast enough so that the overall run is faster. The natural expectations would be that this first step will be fast enough when either

- the task at hand allows us a large increase in the minimum support, or
- there are few ancestors as internal nodes of the hierarchy, since this leads to a small work necessary for the first step.

4.3.2 Our algorithm

So, our algorithm is as follows:

Input: A database DB, a hierarchy J, minimum support σ and increased minimum support σ'.

All frequent itemset of J are calculated using σ'.

All frequent itemsets in $J \cup T$ with minimum support σ are calculated, but pruning all the candidates that do not have a more general itemset declared frequent in the first step.

4.4 Experiments

We have successfully adapted the algorithm R&G to work with taxonomies. We have developed two versions, in order to test the potential improvements of our pruning proposal. This testing has been made only on relatively trivial taxonomies where only one level of internal nodes exists, in an effort to understand the consequences of our proposal.

The algorithm *Jer* (for the catalan word "Jerarquia") implements R&G with implicit enlarged records; of course it prunes the itemsets that contain an item and its ancestors, but no other optimization is made since we are only applying it to two-level hierarchies. The algorithm *JerM* (M is for the catalan word "Millorat", improved) implements the two-step approach with increased frequency threshold, as we have proposed. The fact that both algorithms share many aspects of implementation, makes them suitable to compare empirically both schemes. It is worth to mention that our implementations do not use sampling, although this feature could be easily added.

Now we report the results of some experimentation comparing both algorithms: to what extent the task of finding frequent sets for the internal nodes of the taxonomy is worth its price when we use the results to speed up the task of finding all frequent sets, at least, for very simple two-level taxonomies; also, more specifically, what parameters of the structure of the taxonomy are relevant for this comparison.

Our experiments clearly indicate that the success of our approach depends on the *density* of the taxonomy. Indeed, as a function of the number of children per parent, the algorithm *JerM* performs better or worse that the algorithm *Jer*. In the next tables and figures, we present a small but representative part of the results of our experimental comparisons. They describe the behavior of the running time as we vary two parameters: the density of the taxonomy and the increased threshold required for the support. The reference is algorithm *Jer* working with a support threshold of 6%.

	Density			
algorithm	10	14	25	43
JER (6%)	02:54	02:46	01:24	01:07
JER_M (6%)	02:30	02:37	01:33	01:14
JER_M (6,5%)	02:04	01:54	01:30	01:13
JER_M (7%)	00:55	01:50	01:24	01:11
JER_M (7,5%)	00:48	01:47	01:20	01:12
JER_M (8%)	00:36	01:27	01:14	01:12
JER_M (8,5%)	00:33	01:23	01:09	01:14

Table 3: Total process time (hours:minutes) per density and increased frequency to calculate the frequents sets in the Taxonomy. Correct results are produced with an increased support up to $7,5$.

As we can see from table 3, the time spent in the first step of our algorithm dramatically decreases when the increased support is larger, which was to be expected, and also, when *the density is smaller.* This means that the fewer descendants per parent, the faster.

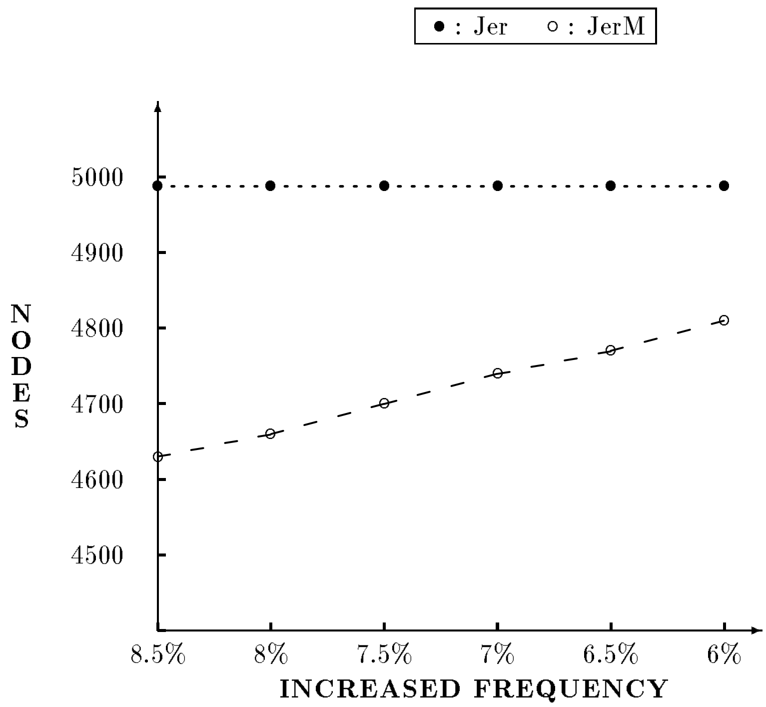

Figure 7: Number of nodes of the hash tree generated for a density of 43 descendents per parent.

It is also interesting to see that the total process time and number of records processed follow the same pattern, which tells us that the speed of the algorithm is strongly dependent on the size of the hashtree and the number of records processed.

In the graphics given we can see two more detailed examples of how the performance of *JerM* is better than that of *Jer* when the density is lower.

We can see that, for a density of 10, *JerM* performs better than *Jer*, but that when the density is increased, *Jer* seems to be a better choice. This is due to the fact that when the density is low, the number of parents is higher. Consequently, their support will not be much higher that that of their children, and thus, precomputing it gives us much more information. To see it clearer, let us compare it with what happens when the density of a parent is high. If parents tend to have many children, their support will tend to be much higher that the children's support, compared to their difference in the former case. Essentially, this means that parents will tend to be frequent easily, and this gives us little chances of pruning itemsets in the second step of the algorithm.

Table 4 compares the number of records processed. It is interesting to note that *JerM* tends to process significantly more records that *Jer*, *even in the cases where the running time ends up being favorable to JerM.* This somewhat surprising behavior is due to the fact that the records processed in the first step,

	Density			
algorithm	10	14	25	43
JER (6%)	153.310	148.580	153.310	156.664
JER_M (6%)	279.989	290.162	291.227	301.485
JER_M (6,5%)	310.047	252.338	294.776	290.427
JER_M (7%)	275.716	244.631	278.908	277.473
JER_M (7,5%)	236.312	239.961	269.171	304.454
JER_M (8%)	209.420	220.534	285.520	287.787
JER_M (8,5%)	210.772	155.491	247.844	272.867

Table 4: Records processed per density and increased frequency to calculate the frequent itemsets of the taxonomy. Good results are produced with an increased support up to $7, 5$.

even though they are a lot, generate a lesser load of handling of the internal data structures, in particular the hash tree. Thus, we are trading more records of low processing overhead for less records more expensive to handle. When the information given by this previous calculation is not useful, the number of hashtree nodes generated by *JerM* is similar to that of *Jer*, and no gain in processing time is achieved.

So, we have seen that our approach is very sensitive to the topology of the taxonomy, basically the density of descendents per parent.

As a final remark, let us note that it looks likely that alternative, more useful information could be computed during the preprocessing of the taxonomy, e.g., refining the frequent sets on the internal nodes adaptively as a function of the actual hierarchy density encountered during the process. All this suggests interesting avenues for further research.

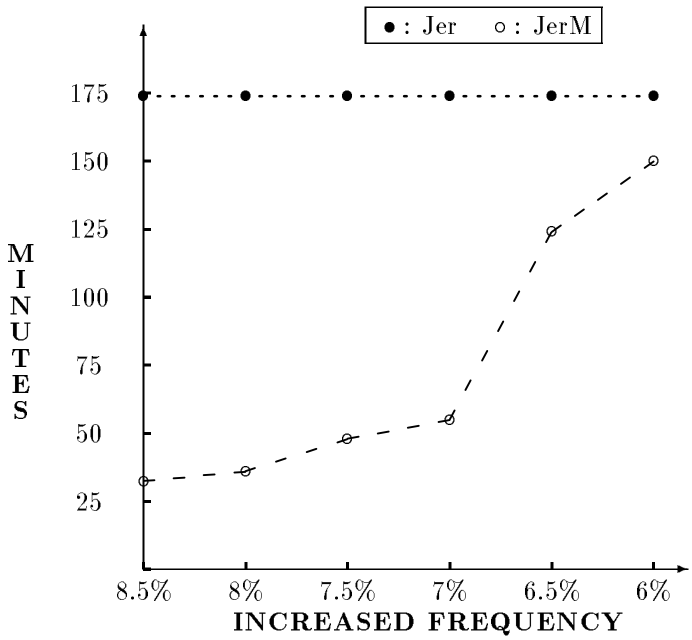

Figure 8: Total process time for a density of 10 descendents per parent.

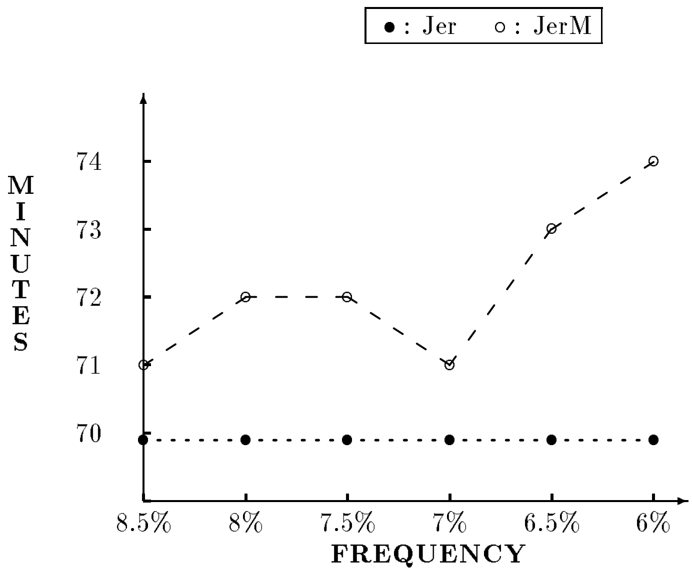

Figure 9: Total process time for a density of 43 descendents per parent. Note that the scale of the Y axis differs from previous figures.

References

[1] R. Agrawal, T. Imilienski and A. Swami. "Database Mining: A Performance Perspective". *IEEE Transactions on Knowledge and Data Engineering*, p. 914-925. 1993.

[2] R. Agrawal, H. Mannila, R.Srikan, H. Toivonen and I. Verkamo. "Fast Discovery of Association Rules". *Advances in Knowledge Discovery and Data Mining*, p. 307-328. AAAI Press, 1996.

[3] R. Agrawal and R. Srikant. "Fast algorithm for mining association rules". *Proceedings of the 20th. VLDB Conference*. 1994.

[4] R. Agrawal and R. Srikant. "Mining Sequential Patterns". *Proceedings of the Int'l Conference on Data Engineering*. 1995.

[5] J. Baixeries. *Hipergrafs transversals etiquetats: Aplicacions*. Projecte Final de Carrera (Graduation Essay). 2000.

[6] S. Brin, R. Motwani, J. Ullman and S. Tsur. "Dynamic Itemset Counting and Implication Rules for Market Basket Data". *Int. Conf. Management of Data*, p. 255-264. ACM Press, 1997.

[7] G. Casas. *Hipergrafs transversals ordenats: Aplicacions*. Projecte Final de Carrera (Graduation Essay) 2000.

[8] C. Domingo, R. Gavaldà and O. Watanabe. "Adaptative sampling methods for scaling up knowledge divery algorithms".To appear in the journal Data Mining and Knowledge Discovery. Preliminary version in Proc. Second International Conference on Discovery Science (DS'99). Springer-Verlag Lecture Notes in Artificial Intelligence 1721 (1999), p. 172-183. A slightly shorter version will be published as a chapter of the book Instance Selection and Construction for Data Mining, Huan Liu and Hiroshi Motoda (eds.), Kluwer Academic Publishers.

[9] D. Gunoppulos, R. Khardon, H. Mannila, H. Toivonen. "Data mining, Hypergraph Transversals, and Machine Learning". *Proc. 16th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems, PODS*, p. 209-216. 1997.

[10] J. Hipp, A. Myka, R. Wirth and U. Güntzer. "A new algorithm for faster mining of generalized association rules". *Principles of Data Mining and Knowledge Discovery*, p.74-82. Proceedings of the second Symposium, PKDD. 1998.

[11] H. Mannila, H. Toivonen and I. Verkamo. "Discovery of frequent episodes in event sequences". *Proc. of the First International Conference on Knowledge Discovery and Data Mining*, KDD. 1995.

[12] K. Matsumoto, T. Hayase and N. Ikeda. "Data mining of generalized association rules using a method of partial-match retrieval". *Discovery Science*. 1999.

[13] S. Parthasarathy, M.J. Zaki, M. Ogihara and S. Dwarkadas. "Incremental and Interactive Sequence Mining". *Proc. of the 8th International Conference on Information Knowledgement*, CIKM (1999). p. 251-258. 2000.

[14] R. Srikant and R. Agrawal. "Mining generalized association rules". *Proceedings of the 21st Int'l Conference on Very Large Databases*, VLDB. p. 407-419. 1995.

[15] H. Toivonen. "Sampling Large Databases for Association Rules". *Proceedings of the 22nd conference on Very Large Databases*, p. 134-145. 1996.