Alcom-FT Technical Report Series
ALCOMFT-TR-01-39

A Faster Algorithm for Finding Frequent Sets *

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

G. Casas-Garriga, J.L Balcázar
Deptartament LSI
Universitat Politècnica de Catalunya
gcasas@lsi.upc.es, balqui@lsi.upc.es

April 5, 2001

Abstract

The association rule discovery problem consists of identifying frequent itemsets in a database, and, then, forming conditional implication rules from them. For this association task, we introduce a new algorithm that reduces substantially both the number of processed transactions and the total computing time. Our algorithmic proposal relies on the philosophy of a current state-of-the-art algorithm, Dynamic Itemset Counting (DIC), which carries out an advanced incorporation of candidates in order to reduce the number of passes through the data. However, this advanced incorporation of candidates only takes place at some previously specified points during the run, which forces DIC to require an extra parameter that balances the number of processed transactions with a certain running time overhead. We propose a way to incorporate candidates in a on-line fashion by maintaining explicitly the so-called negative border of the current maximal frequent sets at each step. The resulting algorithm, Ready-and-Go, is competitive with previous algorithmic proposals, beating them in terms of number of processed transactions and, most importantly, in actual running time. Furthermore, Ready-and-Go turns out to fit perfectly into on-line sampling techniques, which make our algorithm still more efficient. Additionally, in order to show its flexibility, we have adapted it to mine generalized association rules and episodes in sequences, obtaining good results.

*This work is supported by IST Programme of the EU under contract nuber IST-1999-14186 (ALCOM-FT), the EU Working Group EP27150 (Neurocolt II), the Spanish Goverment Grant PB98-0937-C04-04, and CIRIT 1997SGR-00366

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], [2], and [3]. 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 of this task, 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 to finding those itemsets that are frequent enough in the database.

There is a more general problem for this task: 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 [11], [13], [15].

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 of occurrence, that is, the exact moment when that event occurs. 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 are called episodes. Variants of frequent sets algorithms can be used for this purpose (e.g [12], see also [14], [4]).

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 [7], 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, yet controlling the computing time. We 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. Furthermore, we will show how our algorithmic proposal turns out to be structurally addressed to take benefit of on-line sampling techniques (see [9]), which make it still more efficient.

Finally, to explore the flexibility of our proposal, we try and adapt it also to the case of mining episodes from sequences, and to mine generalized association rules in the case that a taxonomy exists on the elements of the database.

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 customer in 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 describe the problem and review two existing solutions. In the next section we present a new proposal to find frequent sets. Then, we experimentally compare our new proposal against previous approaches, obtaining 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* or *frequency* 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","interest" or "conviction" based on ratios of the supports of A, B, $\neg B$ or $A \cup B$, and there are several proposals about how to define them; then those rules having large enough such ratios are deemed potentially useful.

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 performance of the algorithm for the first step, which is by far the most computationally intensive phase, and we do not discuss at all how to define confidence; see [7].

2.2 Previous work on finding frequent itemsets

Since the introduction of the problem of mining association rules ([1], [2], [3]) 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.

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 appear with a frequency that exceeds 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. After this, it counts their support and determines which ones are the frequent 2-itemsets. It continues on 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)** [7] 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

keeps running circularly over the data, counting the support of the current candidates, but stopping at intervals M transactions apart, where M is a user-specified parameter. Therefore, every M processed transactions, DIC algorithm stops counting, makes a generation of all the possible candidates that can be already incoporated into the current pass over the database, and then resumes counting.

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" 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 behaves exactly 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 considerable 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 [6], [8] 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, for very small values of M, Apriori works better, and for large values of M, DIC and Apriori behave computationally similar.

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. Thus, at the time of tackling a concrete frequent sets problem, one is not sure of what is the best choice for M, nor of whether a specific choice will lead to DIC being actually faster than plain Apriori. This is an important

practical limitation for the use of DIC.

3 A new algorithm: Ready and Go

The idea of this algorithm is to try to implement the 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. This can lead to the effect of an on-line incorporation of candidates, that is, candidates will be incorporated as soon as possible.

3.1 Intuition

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 throughout 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. So, 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**, whose support is not being counted, yet some of their immediate subitemsets are candidate itemsets.

In the terminology and notation of [10], this 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, 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 supersets 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 itemset, and the task of computing its support starts at that pass over the database. 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.

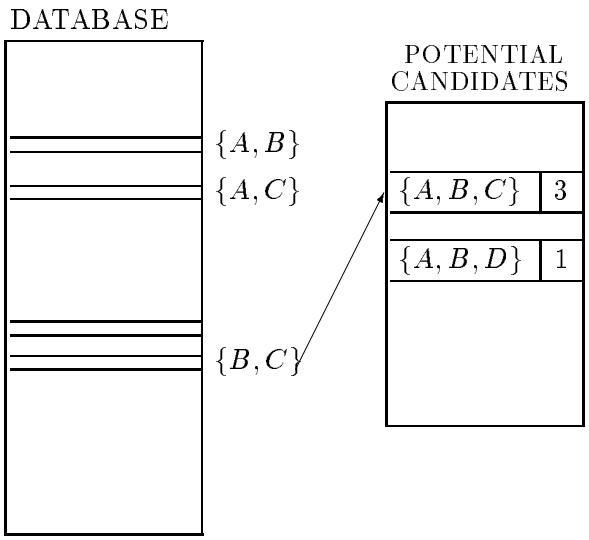

Figure 1: Example of how R&G works.

Example 3.1 *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 superset* $\{A, B, C\}$ *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.*

3.2 The data structures

The implementation of the R&G algorithm requires a Hash Tree, a widely used structure, e.g. [2], [7], that can keep track of many itemsets and support some specific operations on them. When it comes to the structure that can keep track of sets in the current negative border, that is, the potential candidates, the R&G algorithm uses a standard 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, and has an associated counter. 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 immediately incorporated into the

set of candidates.

3.3 The algorithm

The R&G algorithm works as follow:

1. Initialize the candidate itemset collection with all 1-itemsets, and the potential candidate collection with an empty set.

2. Read a new transaction from the database and increment the support of all candidates that appear in the transaction.

3. If a candidate becomes frequent (its support exceeds the user-specified threshold), then notify this situation to all its immediate supersets. This notification consists of incrementing the internal counter of those immediate supersets already located in the potential candidate collection. The other supersets that are not yet in this collection must be added now to it and their internal counter must be set to 1.

4. If the counter of a potential candidate α of size k achieves the value of k, then remove α from the potential candidate collection and add α to the candidate itemset collection.

5. If a candidate has been counted through all the transactions, remove it from the candidate itemset collection.

6. If we are at the end of the transaction file, rewind to the beginning.

7. If any candidate itemsets remain, go to step 2.

3.4 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. Furthermore, we tested our algorithm in real world databases too. We used for this purpose PUMS files, which are Public Use Microdata Samples and contain current census entries. We used the PUMS files for Ohio, Oregon, New York and Texas.

In our experiments R&G has been tested against both DIC and Apriori. As claimed in [7], the best interval M in DIC is close to a value of 1.000 transactions. Thus, this is the value used in all our experiments.

We used an average size of 10 items chosen from 100 items. We focused our experiments in testing the behaviour of R&G as the following two factors vary: *minimum support* and *length of the database*.

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

3.4.1 Varying the minimum support

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

	Apriori	**DIC**	**R&G**
2%	393.045	250.523	249.630
1.5%	491.306	254.523	253.695
1%	491.306	256.523	254.676
0.8%	491.306	266.523	264.007
0.6%	491.306	259.523	257.950
0.3%	589.467	279.523	277.867

Table 1: Total amount of processed transactions in terms of the support.

In particular, in table 1 we can see that R&G processes between 1.000 and 2.000 fewer transactions than DIC, and considerable less transactions than Apriori. This number of saved transactions of R&G with respect to DIC is highly dependent on the size of interval M and on the structure of the data. In general, we can state that the larger the M in DIC, the bigger is the difference in processed transactions between DIC and R&G. In this experiment we are mining synthetic data, which is very homogeneous. This fact produces a uniform incorporation of candidates and consequently, as mentioned in section 2.2.3, DIC and R&G are similar as far as number of processed transactions is concerned.

The number of processed transactions in Apriori depends on the size of the largest candidate incorporated during the run. For example, with a frequency threshold of 0.3%, frequent itemsets were of size 5, and the largest candidates were of size 6; this leads Apriori to make 6 passes through the data (see table 1).

	Apriori	**DIC**	**R&G**
2%	15	13	12
1.5%	22	17	15
1%	29	24	22
0.8%	35	32	29
0.6%	45	41	37
0.3%	125	122	109

Table 2: Time in minutes spent by the tree algorithms when varying the support.

The improvement of R&G regarding the total computing time can be

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

seen in table 2. It is important to highlight that both Apriori and DIC run in relatively similar computing time; nevertheless, when decrementing the support threshold, it appears an important gap between our advantageous algorithm and the other two. The reason of this behaviour can be the number of candidates found out during the run. As this number of candidates becomes larger, the Hash Tree structure grows up, and the generation of candidates every M transactions in DIC, turns out to be more costly. R&G does not suffer from this overhead since the generation of candidates is constant in time. We will confirm this intuition with real world data, where items are usually more correlated, thus forming a larger number patterns.

3.4.2 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.

	Apriori	**DIC**	**R&G**
100.000	491.306	259.523	257.950
250.000	1.228.421	651.369	648.332
500.000	2.457.016	1.301.807	1.299.898
750.000	3.685.726	1.943.291	1.941.000
1.000.000	4.914.741	2.583.897	2.581.855

Table 3: Total number of visited transactions in terms of the length of the database.

	Apriori	**DIC**	**R&G**
100.000	45	41	37
250.000	100	89	77
500.000	190	169	145
750.000	283	248	211
1.000.000	377	328	278

Table 4: Total amount of time in minutes 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 3 and in table 4.

Table 3 shows the regularity of R&G performance. With respect to processed transactions, R&G processes around 2.000 fewer transactions than DIC. This was to be expected since we are again mining synthetic data.

More important is the total computing time of the algorithm. These results can be seen in table 4: for one million transactions, Apriori took over 6 hours, and DIC about 5 and a half, whereas R&G finished after about 4 and a half hours. In general, R&G beat DIC by about 15% 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. This ratio of improvement between both algorithms keeps constant as we enlarge the database; the reason is again the homogeneity of synthetic data: in this experiment, the largest candidate itemset is of size 5 in every run. This means that the number of discovered patterns is constant despite incrementing the length of the database.

3.4.3 Real world databases

In this situation data tends to have a great number of patterns, and so, the candidate itemset collection is expected to be larger than in syntethic databases. So, we varied the support threshold from 80% to 50% in a 66.118-transaction database. These thresholds are assumed to be very high, but we must take into account that we are mining real world databases and this is the natural parametrization. For this experiment we just compared DIC against R&G. Results of this part can be seen in tables 5 and 6.

	DIC	**R&G**
50%	684.807	678.978
60%	553.945	549.299
70%	377.591	374.190
80%	319.473	317.531

Table 5: Processed transactions when varying the frequency threshold

	DIC	**R&G**
50%	291	255
60%	50	47
70%	11	10
80%	3	2

Table 6: Minutes lasted by DIC and R&G when varying the frequency in real databases

Table 5 shows how the number of saved transactions of R&G increase with respect to former experiments. Furthermore, when lowering the

support, the difference in transactions tends to be bigger. So, for example, with 50% of support, the number of saved transactions in R&G is about 6.000.

On the other hand, in table 6 we can see that exists a considerable improvement in time when support with a 50% of support; this fact confirms that, the lower the support in real databases, the more frequent candidates appear and, consequently the larger the improvement of R&G over DIC.

Let us point out that only a small fraction of our large amount of experimentation is explicitly reported here; the trends presented in this section were confirmed by a broad empirical process.

4 Random Sampling for Frequent Sets

We proved that R&G can efficiently reduce the number of processed transactions to discover frequent itemsets; however, the computation bottleneck of any of the above presented algorithms is still the size of the database. An obvious approach to lower the complexity of these algorithms is the application of random sampling.

The usual sampling approach (see, e.g [16]) is to sample randomly a portion of the database of a previously determined size, and then calculate the frequency of the itemsets over the sample using a lower treshold σ', slighty smaller than the user-specified minimum support σ. The most important problem in this *batch sampling* method is to determine the right sample size so that the sample is representative of the whole database. Apart from this, it is likely that this fixed sample size overestimates and that a smaller portion of the data would be enough to extract almost the same statistical conclusions.

An alternative approach to batch methods is *on-line sampling* [9]. In this approach the number of necessary transactions to count the frequency of an itemset can vary adaptatively depending on the situation, that is, it can be determined in an on-line fashion whether the current processed transactions suffice to classify an itemset as frequent or non-frequent. The basic idea of the method is to converge to the real frequency of the itemset while processing transactions. So, the algorithms implementing this method read random transactions one at a time, and try to estimate the real frequency of itemsets according to the transactions processed up to that point.

Both batch and on-line approaches use a reduced threshold σ' during the mining; the consequence of using σ' is that the discovered frequent sets are usually a superset of the frequent sets extracted with deterministic algorithms. However, although in some situations this superset is a good final aproximation, it is possible to remove extra sets by performing a posterior check using the whole database (see [16]).

4.1 On-line R&G

To explore the flexibility of our algorithm, we tried to apply on-line sampling techniques over R&G. Our algorithm fits better than other algorithms into on-line sampling because potential candidates can become candidates at any moment during the run. The adaptativeness of the sampling method helps to achieve this goal efficiently.

Because of the structure of R&G, adapting on-line sampling to our algorithm turns out to be very straightforward. In fact, this on-line fashion helps to create a R&G more efficient: while reading random transactions, each individual candidate can decide its frequency thanks to on-line adaptativeness; once a candidate is declared frequent R&G ideas can be easily applied. We call this resulting algorithm On-line R&G.

Our experiments are oriented to show not only the flexibility of R&G, but also the considerable improvement that on-line random sampling can bring to our algorithm. So, we present in this section two experimental views; first we compare R&G against On-line R&G in case of mining synthetic databases and later in case of mining real databases.

4.1.1 Synthetic databases

In our first experiment, we chosed a 1.000.000-transaction synthetic database (documented in [3]). We tested several values for the minimum support, σ, from 10% down to 5%, and we used a reduced threshold σ' representing a 10% reduction of σ.

The results of our experiments can be seen in table 7 and in figure 2.

	10%	**9%**	**8%**	**7%**	**6%**	**5%**
R&G	2.247.473	2.209.039	2.342.571	2.305.641	2.341.077	2.304.771
On-line R&G	62.372	63.560	63.634	64.153	65.047	66.573

Table 7: Total number of processed transactions

We see that On-line R&G processes considerable less transactions than R&G and this fact causes a high reduction of computing time. Regarding to the number of discovered frequent sets, on-line R&G always finishes with an adjusted superset of the final result: to illustrate this situation, with $\sigma = 7\%$, On-line R&G finds out 100 frequent sets of which only 13 had actually a frequency between 6.3% and 7%. More specific results regarding synthetic databases can be found in [6], [8].

4.1.2 Real world databases

The most interesting application of sampling is when mining real world databases. In this situation, data tends to be non-homogeneous and very

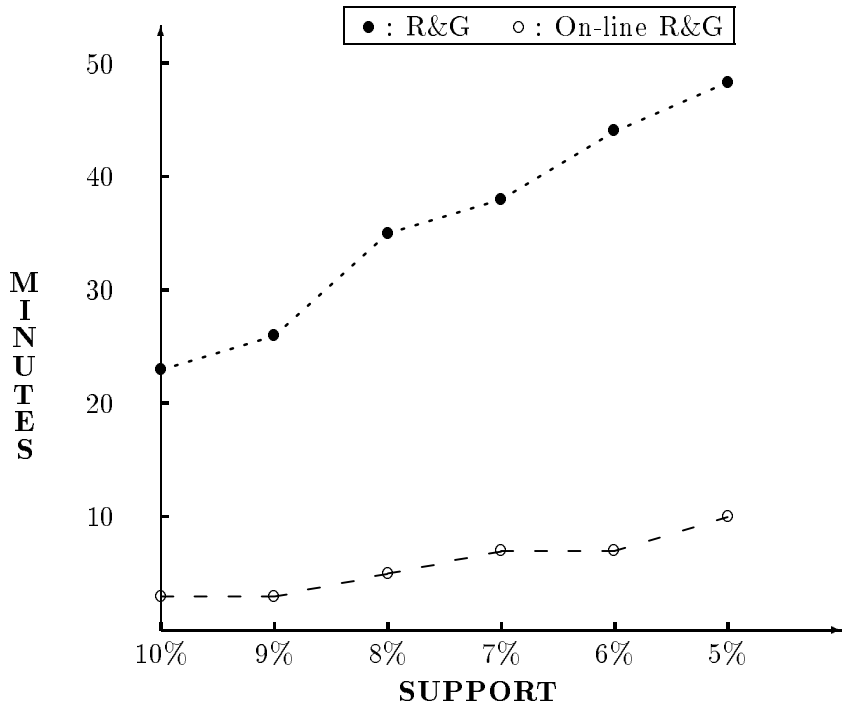

Figure 2: Time spent by two versions of R&G

correlated, and deterministic algorithms, such as DIC or R&G, cannot take as much benefit from the advanced incorporation of candidates.

We used part of U.S census data extracted from PUMS files which are Public Use Microdata Samples. We tried a 66.118-transaction database and we varied the minimum support, σ, from 50% to 80%. Again, we used σ' representing a 10% reduction of σ. Results of processed transactions are shown in table 8.

	R&G	**On-line R&G**
50%	678.978	7.033
60%	549.299	5.393
70%	374.190	4.195
80%	317.531	2.997

Table 8: Total number of processed transactions of a real database

On-line R&G processes less transactions than R&G. Again, this profit in saved transactions allow to get a faster algorithm: for example, for 60% of support R&G needed 47 minutes, while On-line R&G just lasted 2 minutes. More details of these results can be found in [6], [8].

5 Extensions and variations

One of the advantages of R&G is its flexibility: it has been successfully combined with sampling techniques to accelerate the finding of approximate results, and now we try to extend R&G to the problem of finding generalized association rules, and also adapt the algorithm to mine episodes from sequences. Again we prove that our algorithm has a similar adaptability by combining it successfully with these and related ideas.

5.1 Mining Sequences

We consider now 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 3. 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.

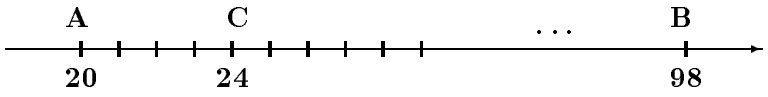

Figure 3: A sequence of events.

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.

Episodes can be described as directed acyclic graphs. Consider, for instance episodes α, β and γ in Figure 4. 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 is usually: $B \rightarrow C$. 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. A more formal description of this problem can be found in [12] and in [5].

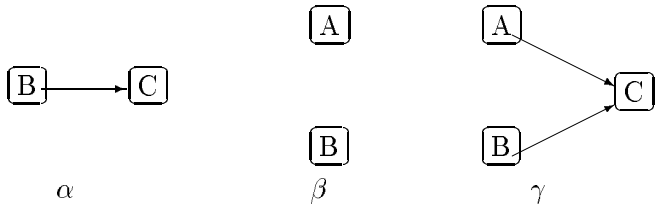

Figure 4: Types of episodes

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.

In the first algorithm in [12], WINEPI, 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. The user defines this fixed window width. The approach consists of applying the Apriori algorithm in sequences as though each window was a transaction with ordered events. 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.

Their second algorithm, MINEPI, is based on *minimal occurrences* of episodes in a sequence. For each frequent episode, the algorithm finds the location of its minimal occurrences: shortest windows on which the episode occurs. The user provides a maximum width of a minimal occurrence and a required number of occurrences, rather than a frequency, since the lengths of the minimal occurrences may vary, and thus a uniform ratio could be misleading.

In [5], we have developed some comparisons of advantages and disadvantages of these mentioned approaches, and proposed yet another form of interestingness for episodes, in which the user indicates a maximum distance between events to be part of the same episode, instead of a window width; and an according notion of frequency. The benefits include the avoidance of separate but overlapping frequent episodes, not losing any of the episodes found by WINEPI, and a uniform notion of frequency that allows sampling.

Finally, let us point out that an alternative approach for sequences was introduced by [4] and also studied by [14]. 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. This is similar to concentrating on serial episodes and, with some easy tricks

on the manipulation of timestamps, concatenating together all data into a single sequence. Thus, we choose to focus on the problem definition first mentioned.

5.1.1 Sequential Ready&Go

In this case in which the algorithm must look for episodes, the basic ideas of R&G can be applied. To follow the same structure, this new algorithm, called Sequential 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 when α has all its parts of size $k-1$ frequent. 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. Which subepisodes have to be considered parts of α depends on whether α is serial, parallel, or hybrid. A more detailed discussion appears in [5], in which we have also developed a strategy for finding simultaneously serial and parallel episodes.

The facts that are relevant here are that, first, given a proper definition of parts of an episode, and of a measure of interestingness, the Ready and Go algorithm can be applied to the mining of serial or parallel episodes in sequences with no particular difficulties, yielding an efficient software that we have successfully tested with both syntetic and real data of up to 8 Mbytes (raw data).

5.2 Generalized association rules

We have also applied the R&G algorithm to the task of finding frequent sets in case of having a hierarchical taxonomy on the items of the database; and we have used it to explore the interest and cases of success of a heuristic proposed in [5].

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 the following taxonomy: canned beef is canned meat, and canned meat is meat; on the other hand, apples are fruits. Then, 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.

5.2.1 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.

In fact, this simple framework is used in several algorithms, such as in the algorithms *Basic* [13] and *Prutax* [11]. This framework has two good properties: first, it is conceptually easy to understand and it is intuitive; and second, any frequent sets algorithm can be applied to the enlarged database. Apriori is the usual choice, but other algorithms have been successfully adapted. Of course, the enlarged transactions can be programmed to be made "on demand", so that the enlarged database never actually exists in the system.

We have followed this approach with R&G. However, this approach also has drawbacks: the main one is that the hash tree of the candidate sets increases substancially.

One new proposal was made by the present authors in [5]. It can be summarized as follows: the algorithm computes first all frequent sets formed *only* by internal nodes of the taxonomy, and then uses this information to avoid unnecessary generation of candidates on the complete database. Additionally, the possibility of using different frequency thresholds for the two phases leads to interesting consequences [5].

This approach has a sort of "gambling" component that makes it

doubtful, since it is unclear whether the extra pruning in the second phase would make up for the time spent in the first phase. In addition, it is unclear how much we can increase the frequency threshold for the first phase without making useless the whole task, and how much we gain from each small additional increase.

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. Our experiments are being conducted on synthetic databases obtained using the IBM generator, together with our own hierarchy generator to construct the taxonomy on it. Our generator allows us to control explicitly the branching factor of the hierarchy (i.e. average number of offspring nodes per internal node) so that this factor can be taken into account in the experiments.

For both variants, extensive testing is being conducted on databases of length of 100.000 records, with average record sizes of 10 items, combined with two-level taxonomies consisting of parents with a density of children varying from 10 to 43 children per parent. Our findings indicate that Ready-and-Go takes about 1 to 3 hours of CPU time on a Pentium II with 200Mhz and 256Mb of main memory. This suggests that the algorithm can be suitable for real-world problems of realistic size.

Specific data regarding this experimentation can be found in [5]; our main point in discussing this ongoing work here is essentially, as in the previous subsections, that our algorithm Ready-and-Go is not only competitive with state-of-the-art techniques with regard to processing time, but also flexible enough to allow its adaptation to more general problems, such as the cases of sequences or taxonomies.

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. Srikant, 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, G. Casas-Garriga and J.L. Balcázar. "Frequent sets, sequences, and taxonomies: new, efficient algorithmic proposals". Report number: LSI-00-78-R. 2001.

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

[7] 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.

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

[9] C. Domingo, R. Gavaldà and O. Watanabe. "Adaptative sampling methods for scaling up knowledge divery algorithms".To appear in *Data Mining and Knowledge Discovery.* See also: Proc. Second International Conference on Discovery Science (DS'99), p.172-183. 1999.

[10] D. Gunopulos, 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.

[11] 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. Proc. 2nd Symposium, PKDD. 1998.

[12] 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.

[13] K. Matsumoto, T. Hayase and N. Ikeda. "Data mining of generalized association rules using a method of partial-match retrieval". Proc. 2nd Int. Conf. on Discovery Science (DS'99), p.160-171. 1999.

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

[15] 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.

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