Sequential Pattern Mining Algorithms: Trade-offs between Speed and Memory

Cláudia Antunes and Arlindo L. Oliveira

Instituto Superior Técnico / INESC-ID,
Department of Information Systems and Computer Science,
Av. Rovisco Pais 1,
1049-001 Lisboa, Portugal
{claudia.antunes, arlindo.oliveira}@dei.ist.utl.pt

Abstract. Increased application of structured pattern mining requires a perfect understanding of the problem and a clear identification of the advantages and disadvantages of existing algorithms. Among those algorithms, pattern-growth methods have been shown to have the best performance when applied to sequential pattern mining. However, their advantages over apriori-based methods are not well explained and understood. Detailed analysis of the performance and memory requirements for these algorithms shows that counting the support for each potential pattern is the most computationally demanding step. Additionally, the analysis makes clear that the main advantage of pattern-growth over apriori-based methods resides on the restriction of the search space that is obtained from the creation of projected databases. In this paper, we present this analysis and describe how apriori-based algorithms can achieve the efficiency of pattern-growth methods.

1 Introduction

The rapid growth of the amount of stored digital data and the recent developments in data mining techniques, have lead to an increased interest in methods for the exploration of data, creating a set of new data mining problems and solutions. *Frequent Structure Mining* is one of these problems. Its target is the discovery of hidden structured patterns in large databases. Sequences are the simplest form of structured patterns.

In the last decade, a number of algorithms and techniques have been proposed to deal with the problem of sequential pattern mining. The main approaches to sequential pattern mining, namely apriori-based and pattern-growth methods, are being used as the basis for other structured pattern mining algorithms. However, and despite the fact that pattern-growth algorithms have shown better performance in the majority of the situations, its advantages over apriori-based methods are not sufficiently understood.

In this paper, we study the problem of sequential pattern mining, in order to explain the main reasons why pattern-growth methods outperform apriori-based ap-

proaches. However, a fair evaluation of the methods requires that they have exactly the same goals, which is not true for the best-known algorithms, *GSP* and *PrefixSpan*. In order to accomplish our goal, we use a generalization of *PrefixSpan* (*GenPrefixSpan*) [2] that deals with gap constraints, and maintains the pattern-growth philosophy. From this analysis, we conclude that apriori-based methods may become as efficient as pattern-growth methods under specific conditions, and present a new apriori-based algorithm – *SPaRSe* (Sequential PAttern mining with Restricted SEarch) that uses both candidate generation and projected databases to achieve higher efficiency for high pattern density conditions.

The rest of the paper is organized as follows: section 2 exposes and formalizes the problem, presenting its comparison to *Frequent Itemset Mining* problem, and an analysis of apriori-based and pattern-growth methods when using gap constraints. Section 3 describes a new apriori-based algorithm – *SPaRSe*, which implements new procedures for support based pruning, candidate generation and candidate pruning. Section 4 describes a complete performance study over synthetic and real-world datasets, used to demonstrate our claims and to discuss the advantages and disadvantages of each approach. Section 5 finishes, drawing the most relevant conclusions.

2 Sequential Pattern Mining

Sequential Pattern Mining algorithms address the problem of discovering the existent maximal frequent sequences in a given database. Algorithms for this problem are relevant when the data to be mined has some sequential nature, i.e., when each piece of data is an ordered set of elements, like events in the case of temporal information.

The problem was first introduced by Agrawal and Srikant [1], and since then the goal of sequential pattern mining is to discover all frequent sequences of itemsets in a dataset. In particular, an *itemset* is a non-empty subset of elements from a set C, the item collection, called *items*. In this manner, an itemset represents the set of items that occur together. The itemset composed of items *a* and *b* is denoted by (*ab*).

A *sequence* is an ordered list of itemsets. A sequence is *maximal* if it is not contained in any other sequence. A sequence with *k* items is called a *k*-sequence. The number of elements (itemsets) in a sequence *s* is the length of the sequence and is denoted by |*s*|. The i^{th} itemset in the sequence is represented by s_i and the set of considered sequences is usually designated by database (DB), and the number of sequences by database size (|DB|).

A subsequence *s*' of s is denoted by $s' \subseteq s$. Formally, a sequence $a=<a_1 a_2 ... a_n>$ is a *subsequence* of $b=<b_1 b_2 ... b_m>$, if there exist integers $1 \le i_1 < i_2 < \ldots < i_n \le m$ such that $a_1 \subseteq b_{i_1}$, $a_2 \subseteq b_{i_2}$, …, $a_n \subseteq b_{i_n}$.

One of the simplest constraints applied in the discovery of sequential patterns is the gap constraint. It consists on imposing a limit in the maximum distance between two consecutive elements in the sequence. This simple constraint is very useful to reflect the impact of some item on another one, in particular, when each transaction occurs at a particular instant of time. In this manner, it is possible to specify that an

event has greater impact on near events than on distant ones. When using gap constraints, the notion of contained in has to be adapted: a sequence $a=<a_1a_2...a_n>$ is a *δ-distance subsequence* of $b=<b_1b_2... b_m>$, if there exist integers $1\leq i_1<i_2<\ldots<i_n\leq m$ such that $a_1\subseteq b_{i1}$, $a_2\subseteq b_{i2}$, …, $a_n\subseteq b_{in}$.and $i_k-i_{k-1}\leq\delta$. Sequence *a* is a *contiguous subsequence* of *b* if *a* is a 1-distance subsequence of *b*, i.e., the items of *a* can be mapped to a contiguous segment of *b*. Note that a contiguous subsequence is a particular case of δ-distance subsequence and is the most restrictive notion of subsequence. A δ-distance subsequence s' of s is denoted by $s' \subseteq_{\delta} s$. Using δ=1 eliminates the possibility of having gaps between consecutive items. In the rest of this paper this is designated by gap=0.

2.1 Problem Analysis

What makes this problem more challenging than frequent itemset mining? It is obvious that frequent itemset mining is just a particular case of sequential pattern mining, since frequent itemsets are a particular case of sequential patterns – *1*-sequential patterns. Sequential pattern mining requires, beside the discovery of frequent itemsets, the arrangement of those itemsets in sequences and the discovery of which of those are frequent.

To understand why there exists a significant increase in the number of potential patterns, assume that there is a database to be mined with the minimum support threshold set to σ and with $n=|C|$ different items in the item collection, *C*. The goal of frequent itemset mining is to find which itemsets are frequent from the $|I|$ different possible existent itemsets, where *I* is the powerset of *C*, and its value is given by equation (1).

$$|I| = \sum_{j=1}^{n} \binom{n}{j} - 1 = 2^n - 1 \tag{1}$$

To understand the sequential pattern mining problem, let's begin by considering that the database has sequences with at most *m* itemsets and each itemset has at most one item. In these conditions, there would be n^m possible different sequences with m itemsets and

$$\sum_{k=1}^{m} n^k = \frac{n^{m+1} - n}{n-1} \tag{2}$$

different arbitrary length sequences. Similarly, if each itemset has an arbitrary number of items, there would exist S_m possible frequent sequences with *m* itemsets, with the value of S_m given by equation (3).

$$S_m = |I|^m = (2^n - 1)^m \tag{3}$$

And, there would exist S sequences in general, as in equation (4).

$$S = \sum_{k=1}^{m} (2^n - 1)^k = \frac{(2^n - 1)^{m+1} - 2^n - 1}{2^n - 2} = \Theta(2^{nm}) \tag{4}$$

Indeed, the number of different items and the average length of frequent sequences are tightly connected: a large number of items in a short sequence may imply a reduced number of frequent patterns, since the probability of the generality of items has to be small. In this way, algorithms will run efficiently. The opposite situation is a large sequence with a reduced number of items, where the probability of each element to occur in a large number of sequences is high. This leads to the existence of many patterns, and consequently a large amount of processing time. The concept of database density quantifies this relationship. The database *density* (ρ) is the ratio between the number of existing patterns (F) and the number of possible sequences S.

In practice, the database density depends on the support considered. The density is higher when the minimum support thresholds are low, since there are a larger number of frequent sequences. Another parameter that has impact on the density is the set of items existing in the database. On one side, a large number of items allows for a potentially large number of different sequences. Since with many distinct sequences their probability to be frequent is low, there will exist (all other conditions being equal) a smaller number of patterns and the database density will be low. On the other side, a reduced number of items generates a smaller number of potential sequences, which will be more frequent in the database, increasing the number of patterns and consequently the density of the database. Despite the potentially large number of sequences (expression (4)), only a small fraction will be, in general, supported by the database. In particular, there can only exist $1/\sigma$ sequences of length m that are frequent. Given this discrepancy between the number of different sequences and frequent ones, the difficulty of the data mining process resides in figuring out what sequences to try and then efficiently finding out which of those are frequent [1].

2.2 Analysis of Algorithms

There are several algorithms for mining sequential patterns. AprioriAll [1], GSP [7], SPADE [8], PrefixSpan [5] and Spam [3] are just the simpler ones (simple in the sense that they not introduce complex constraints in the mining process). From these, GSP and PrefixSpan are the best-known algorithms, and represent the two main approaches to the problem: apriori-based and pattern-growth methods. Next, we will describe both approaches and compare their advantages and disadvantages.

Apriori-based Approaches. *GSP* follows the candidate generation and test philosophy. It begins with the discovery of frequent *1*-sequences, and then generates the set of potentially frequent (*k+1*)-sequences from the set off frequent *k*-sequences (usually called candidates). The generation of potentially frequent *k*-sequences (*k*-candidates) uses the frequent (*k-1*)-sequences discovered in the previous step, which may reduce significantly the number of sequences to consider at each moment. Note that to decide if one sequence *s* is frequent or not, it is necessary to scan the entire database, verifying if *s* is contained in each sequence in the database.

In order to reduce its processing time, *GSP* adopts three optimizations. First, it maintains all candidates in a hash-tree to scan the database once per iteration. Second, it only creates a new *k*-candidate when there are two frequent (*k-1*)-sequences with the prefix of one equal to the suffix of the other. Third, it eliminates all candidates that have some non-frequent maximal subsequence. By using these strategies, *GSP* reduces the time spent in scanning the database, increasing its general performance. In general, apriori-based methods can be seen as breath-first traversal algorithms, since they construct all k-patterns simultaneously. Note that, at each step *GSP* only maintains in memory the already discovered patterns and the *k*-candidates.

Pattern-growth Methods. Pattern-growth methods are a more recent approach to deal with sequential pattern mining problems. The key idea is to avoid the candidate generation step altogether, and to focus the search on a restricted portion of the initial database. *PrefixSpan* is the most promising of the pattern-growth methods and is based on recursively constructing the patterns, and simultaneously, restricting the search to projected databases. An α-projected database is the set of subsequences in the database, which are suffixes of the sequences that have prefix α. At each step, the algorithm looks for the frequent sequences with prefix α, in the corresponding projected database. In this way, the search space is reduced at each step, allowing for better performances in the presence of small support thresholds. In general, pattern-growth methods can be seen as depth-first traversal algorithms, since they construct each pattern separately, in a recursive way.

As pointed in [2], when a gap constraint is used, neither *PrefixSpan* nor *PrefixGrowth* [6] can be applied directly. The generalization proposed – *GenPrefixSpan*, is based on the redefinition of the method used to construct the projected databases. Instead of looking only for the first occurrence of the item, every occurrence is considered.

Comparison. In order to understand and identify what are the most time consuming operations of each algorithm, we have performed a profiling study, recording the total time spent by the main steps of each algorithm. Both *GSP* and *GenPrefixSpan* were executed in a set of datasets with several different values for minimum support.

As other pattern-growth methods, *GenPrefixSpan* generally outperforms *GSP*, and has much better results for low minimum threshold support values. In order to understand why this happens, let us analyze the time spent in each step of *GSP* when using low minimum support values. We have considered the two main steps of *GSP*: candidate generation (which includes the initial step, where frequent 1-sequences are discovered; the procedure that defines the sequences potentially frequent and the procedure that eliminates some of the candidates) and candidate test (which corresponds to the support pruning and consumes almost the totality of processing time).

Table 1 shows that the support-based pruning procedure consumes almost the totality of processing time (the experiments are described in the last section). For *GenPrefixSpan*, the relative results are quite different: the processing time spent in scanning the database is approximately 50% and uses much less time than *GSP* for low minimum support values.

Table 1 – Processing times for GSP and GenPrefixSpan

	GSP					GenPrefixSpan				
sup	Candidate Generation		Candidate Test		Total	Find Elements		Create ProjDB		Total
50%	0,01s	0%	3,19s	100%	3,19s	0,88s	50%	0,89s	50%	1,78s
40%	0,01s	0%	4,90s	100%	4,91s	1,33s	53%	1,16s	47%	2,49s
33%	0,02s	0%	9,08s	100%	9,10s	2,01s	55%	1,67s	45%	3,68s
25%	0,02s	0%	17,32s	100%	17,34s	3,40s	55%	2,74s	45%	6,15s
10%	1,26s	1%	157,63s	99%	158,89s	18,71s	58%	13,41s	42%	32,13s

Since both methods spend a large percentage of time scanning the database, what makes *GenPrefixSpan* much faster than *GSP*? The answer lies in the reduction of the search space. In fact, at each recursion step, *GenPrefixSpan* usually scans a smaller database, since the α-projected database has more sequences than the $\alpha\beta$-projected database.

3 The *SPARSE* Algorithm

The results of this analysis lead us to analyze the possibility of applying a search restriction to apriori-based methods. *SPaRSe* (Sequential Pattern mining with Restricted Search) is a new algorithm, which combines the candidate generation and test philosophy with the restriction of the search space obtained from the use of projected databases. It acts iteratively like apriori-based algorithms: after discovering the frequent elements, it looks for patterns with growing length at each step. It finishes when there are no more potential frequent patterns to search. The key idea is to maintain a list of supporting sequences for each candidate, and to verify the existence of support only in the subset of sequences that are super-sequences of both generating candidates, in a way similar to SPADE [8].

Fig. 1 describes the main procedure of *SPaRSe*. Note that it is identical to the main procedure of *GSP*, since the `patternDiscovery` procedure agregates the functionalities of `candidateGeneration`, `candidatePruning` and `supportPruning` in *GSP*. The difference to *GSP* is the fact that *SPaRSe* generates and tests each candidate separately. Procedure `satisfies` counts the support for one candidate and returns true if it is frequent and false otherwise. This is similar to the behavior of `supportPruning` in *GSP*.

However, what makes *SPaRSe* more than a variant of *GSP*, is the restriction of the search space in a way similar to *PrefixSpan*: it associates each frequent discovered pattern with the set of sequences where it appears. This set is called the support database. In this manner, it is possible to count the support of a new candidate, only in the intersection of the support databases of its parents. Note that the anti-monotonicity property implies that if a sequence does not support a pattern, then it could not support any of its super-patterns. When the support of a candidate is counted, only the potential support sequences are scanned.

```
SPaRSe (DB, minsup, δ)
  L1←{frequent 1-sequences}
  for (k=2;Lk-1≠∅;k++) do
    patternDiscovery(Lk-1,DB,minsup,δ,k)
    k ← k+1
return ∪k Lk
```

```
patternDiscovery(Lk-1,DB,min_sup,δ, k)
  Lk ← ∅
  for each s∈Lk-1 do
    for each t∈Lk-1 do
      c←createNewCandidate(s, t, k)
      if possibleFrequent(c.sequence,
                          Lk-1, k, δ)
         ∧ satisfies(c,min_sup,δ)
        Lk←Lk∪{c}
return Lk
```

```
createNewCandidate(a, b, k)
  c←join(a.sequence,b.sequence,k)
  c.supDB←c1.supDB ∩ c2.supDB
return c
```

```
join (s, t, k)
  if (∀1<n<k-2: sn+1=tn)
    return s1…sk-1tk-1
```

```
possibleFrequent(s,Lk-1,k,δ)
return(~∃t⊆δs∧|t|=k-1∧t∉Lk-1)
```

```
satisfies(c, min_sup, δ)
  s← c.sequence
  nr←0
  for each t∈c.supDB do
    if s⊆δt nr←nr+1
return (nr≥sup)
```

Fig. 1. SPaRSe pseudocode

In *SPaRSe* a pattern is not only a sequence in itself, but it contains the information that lists the sequences where it occurs, which corresponds to its support database. This simple modification justifies the new procedure for generating candidates – createNewCandidate. This simple inclusion allows for constraining the search considerably, improving the global average performance. However, maintaining support databases for each discovered pattern, and for every candidate of length *k* is expensive in terms of memory.

A simple way to minimize this problem is to use an array of bits to represent the support database. Note that in *GenPrefixSpan* an α-projected database uses more memory (since it also keeps the sequence identification and the index of the α occurrence). However, *GenPrefixSpan* is a depth-first traversal algorithm, which avoids having all projected databases in memory at the same time. In the case of breath-first traversal algorithms, as *SPaRSe*, the solution is to redesign the candidate generation and test procedure: instead of generating all candidates at once and then testing them, it is possible to generate and test them one by one, minimizing the memory consumption, which explains the design of the patternDiscovery procedure. Note that with this change, it does not make sense to use sophisticated data structures, as hash-trees, to count the support for each candidate. Usually, as has been said, apriori-based algorithms use hash-trees to store all candidates, and scan the database once to count the support for all candidates. Generating and testing each candidate separately does not require the use of these techniques.

However, for very low support thresholds *SPaRSe* does not work better than *GenPrefixSpan*, spending long times in the candidate generation and pruning. Remember that apriori-based methods generate k-sequence candidates by joining two (*k-1*)-patterns, when the prefix of one is equal to the suffix of the other. This operation may consume a considerable amount of time when there are many frequent patterns.

This happens, since for every pattern it is necessary to verify which patterns have a prefix equal to its suffix.

To improve the generation of *k*-candidates, *SPaRSe* stores all (*k-1*)-patterns in a hash-tree. Fig. 2 illustrates this data structure when storing the different combinations of two elements. When the number of items is large and the database is sparse it is useful to use the same leave to store sequences with different prefixes, justifying the use of a hash-tree instead of a suffix-tree.

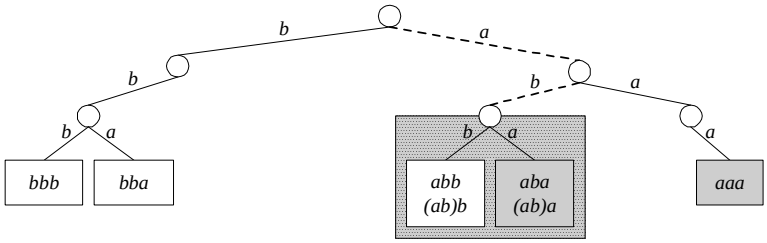

Fig. 2. Hash-trees for candidate generation and pruning

In order to generate a new candidate with length k, *SPaRSe* does not need to inspect every (k-1)-candidate. The algorithm takes the suffix of each candidate *s* and follows its path in the hash-tree. The reached sub-tree contains the sequences that may match with *s* to generate a new candidate. Now it is only necessary to verify if they really match with *s*, and then generate new candidates.

Consider for example the sequence *a*(*ab*). Following the path of its suffix <*ab*> in the hash-tree, we discover the sequences that may match with it – *abb*, (*ab*)*b*, *aba* and (*ab*)*a*. Note that only (*ab*)*b* and (*ab*)*a* are really appropriate to join with it, and generate two new candidates: *a*(*ab*)*b* and *a*(*ab*)*a*. Fig. 2 shows the followed path with a doted line, and the possible matching sequences in the sub-tree inside the dotted box. By avoiding testing if any two patterns match, *SPaRSe* improves its performance by 50%, for low support thresholds.

Finally, we have considered a last improvement – the use of a hash-tree to implement candidate pruning. The key idea of candidate pruning is to eliminate candidates that cannot be frequent, as stated before. However, verifying if every maximal subsequence is frequent for every candidate may be prohibitive, especially when low support thresholds are used. Like candidate generation, this procedure may use a hash-tree to identify the potential frequent patterns. Consider again the hash-tree and the candidate (*ab*)*aa*. It has three maximal subsequences *aaa*, *baa* and (*ab*)*a*. Although the first and second ones are frequent patterns (presented in shadowed boxes), the third one is not, since it is not stored in the hash-tree. Looking for the subsequence's paths in the hash-tree reduces significantly the time needed to make this discovery.

In summary, *SPaRSe* is an apriori-based algorithm, which follows the candidate generation and test philosophy. It has three fundamental differences to *GSP*: it generates and tests one candidate at a time, it uses *support databases* to count the support for each candidate; and it uses a hash-tree to store frequent patterns. These improvements directly contribute to accelerate the candidate generation and pruning procedures.

In the next section, it is shown how *SPaRSe* and *GenPrefixSpan* algorithms deal better with datasets of different characteristics, and that either one of them may represent the best choice for a particular application.

4 Experimental Results

The comparison of sequential pattern mining algorithms over a large range of data characteristics, such as different support thresholds, dataset sizes and sequence lengths, has been done by several authors (see for instance [2], [7], [8] or [5]). However, as stated in section 2, the results depend on the dataset density, and to the authors best knowledge, there has been no study about the performance of sequential pattern mining algorithms in dense datasets. Our goal in this section is to understand the impact of those characteristics in the algorithms' performance. In order to do that, we compare the performance of *GenPrefixSpan*, *SPaRSe*, and *GSP*, on several distinct datasets, considering all the enumerated characteristics. The performance of *GSP* only serves as a reference line to the performance of the other two algorithms, since the execution times are generally much larger than the execution times of the other algorithms. Neither *PrefixSpan* nor *SPAM* [3] could be used, since they do not deal with gap constraints.

To perform the study over a large range of different characteristics, we used the standard synthetic data set generator from IBM Almaden. The datasets used in these experiments were generated maintaining all, except one, of the parameters fixed, and exploring different values for the remaining parameter. In general, the datasets contain 10.000 sequences (Parameter D of the generator set to 10), with 10 transactions each on the average (C=10). Each transaction has on the average 2 items (T=2). The average length of maximal patterns is set to 4 (S=4) and maximal frequent transactions set to 2 (I=2). These values were chosen in order to follow closely the parameters usually chosen in other studies. The values for different sequential patterns (Ns) and transactional patterns (Ni) were also chosen similarly, set to 5.000 and 10.000, respectively. The computer used to run the experiments was a Pentium M 1GHz with 768MB of RAM. The operating system used was Windows XP. The datasets were maintained in main memory during the algorithms processing, avoiding hard disk accesses. The next subsections present the performance results achieved using datasets with different densities, followed by the studies on different support thresholds and different gap values. The section finishes with the scalability studies.

Performance. The behavior of both algorithms is somehow different for different levels of density. As can be observed in Fig. 4(a), *GenPrefixSpan* achieves better results for sparse datasets, but shows performances similar to the ones shown by *SPaRSe* for dense datasets. The main reason for this difference is that *SPaRSe* does not waste so much time generating infrequent candidates for dense datasets. Since there are more patterns, both algorithms have to generate a similar number of sequences, reducing the difference between their processing times. In terms of memory consumptions, as stated before, *GenPrefixSpan* consumes more memory

than *SPaRSe*, since it has to maintain multiple indexes for the same sequence and the corresponding pattern position for each occurrence. The results show that both algorithms consume more memory in processing dense datasets, since the number of patterns is higher. The different values for density were achieved by varying the number of different items in the dataset from ten to one thousand (N∈{10, 20, 30, 40, 50, 100, 500 and 1.000}).

For different minimum support thresholds, the results are consistent. SPaRSe equals GenPrefixSpan in dense situations and shows worse results than for sparse datasets (Fig. 3(b)).

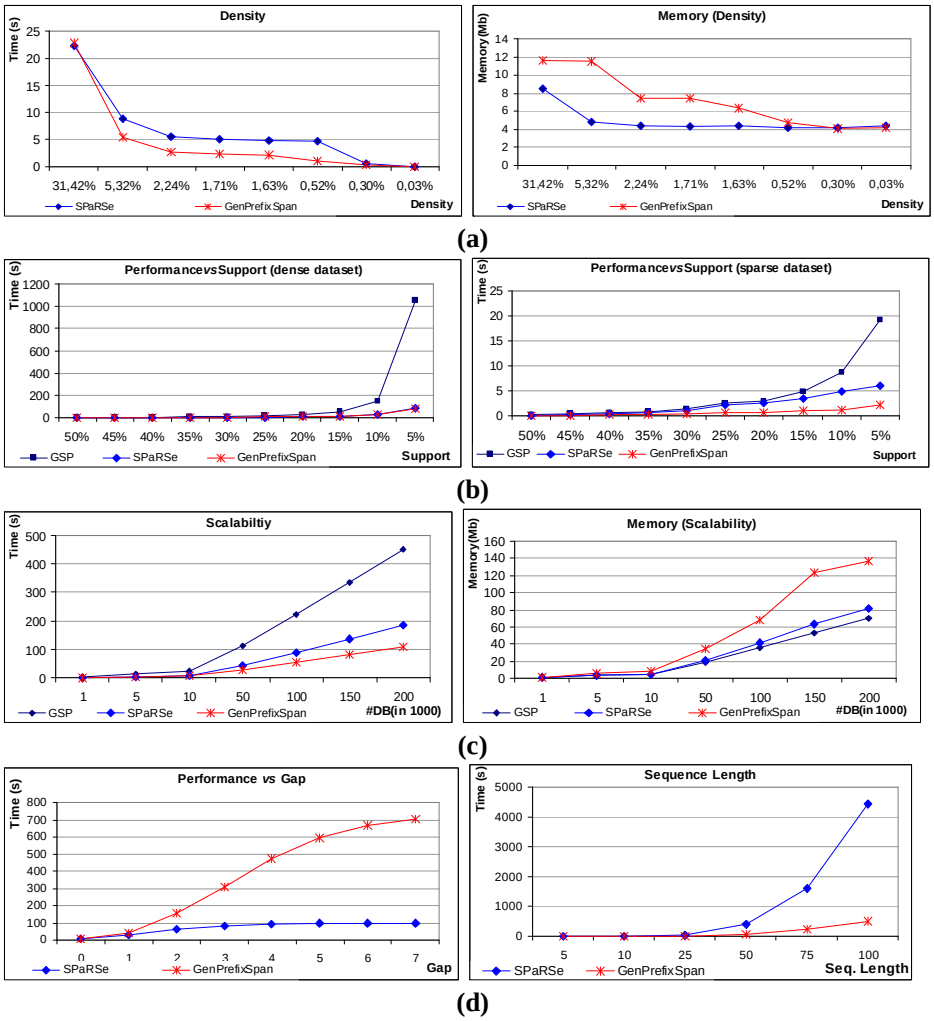

Fig. 3. (a) Performance and memory requirements *vs* dataset densities; (b) Performance *vs* support (dense and sparse datasets); (c) Performance and memory requirements *vs* dataset sizes; (d-left) Performance *vs* gap constraints; (d–right) Performance *vs* sequence length

It is interesting to note that the execution times in sparse datasets are about ten times faster than in dense datasets, for all compared algorithms. These results show that a great part of the efficiency of GenPrefixSpan is due to its levels of memory consumption. In several other experiments, conducted in machines with less available memory, the results were slightly different, with GenPrefixSpan showing worst results than SPaRSe for dense datasets. However, in the presence of machines with not much memory (say 250Mb) the results are again worst for SPaRSe. In fact, since GenPrefixSpan works in a depth-first manner, it is able to manage hard-disks access in a more efficient way.

When comparing the algorithms for different gap constraints, the results are considerably different (Fig. 4(d-left)). GenPrefixSpan is worst than SPaRSe for less restrictive gaps. In fact, while SPaRSe must scan the entire sequences for finding a pattern (even if it is not present in the sequence), GenPrefixSpan only has to look for the items in the positions near to the already discovered pattern prefix. When gap is set to zero, GenPrefixSpan only has to look at the next position, reducing the amount of time needed in scanning the dataset. Furthermore, for longer gaps, the number of sequences in the projected database increases, which also contributes to reduce its performance.

Scalability. Since the most time consuming operation is scanning the database, the results achieved by algorithms for bigger datasets are not surprising. All algorithms present worst behaviors for large datasets, but with slightly different patterns of growth (Fig. 4(c)). The results show that *SPaRSe* and *GenPrefixSpan* present a considerably better performance for very large databases (larger than 10 thousand sequences) than *GSP*. It is interesting to see that *GenPrefixSpan* consumes much more memory than apriori-based algorithms. This difference in memory consumption is due to the creation of projected large databases, since *GenPrefixSpan* has to maintain multiple indexes for the same sequence and needs to store the pattern position for each occurrence, wasting more memory than *SPaRSe*. This is clearly most notorious for larger datasets.

Another important factor in the performance of sequential pattern mining algorithms is the average length of sequences. In order to evaluate different situations, the generated datasets include sequences with different numbers of transactions. Indeed, the sequence length influences the time consumed when looking for each frequent candidate. For long sequences (more than 25 itemsets), the probability of supporting every element is very high. In this manner, SPaRSe is not able to reduce the search space (since the support databases approximately maintain the original size) and its candidate pruning does not eliminate a significant number of candidates. On the other side, GenPrefixSpan only has to look for the next position, efficiently dealing with long sequences (Fig. 3(d-right)).

In summary, the experiments reveal essentially two aspects: GenPrefixSpan outperforms *SPaRSe* in sparse datasets, mainly due the time spent on candidate generation by *SPaRSe*, but they show similar performances on dense datasets; and *GenPrefixSpan* consumes much more memory than *SPaRSe* and *GSP*. The results achieved from the analysis of real-world datasets, confirm the differences on the presence of

dense and sparse datasets (due to space limitations, they are not presented here). Among our experiments, we have applied both algorithms to mine web-logs (very sparse datasets), sequences corresponding to retail customers acquisitions and to mine student's curricula (very dense datasets).

5 Conclusions

In this paper, we analyze the problem of sequential pattern mining in detail. After describing the best-known approaches to this problem (apriori-based and pattern-growth methods), we show that apriori-based algorithms can be optimized to match the execution times of pattern-growth methods. SPaRSe is an optimization of GSP that achieves those goals.

This paper also presents a detailed discussion of the advantages and disadvantages of both approaches (apriori-based and pattern-growth methods) conduced by comparing the performance of *SPaRSe* and *GenPrefixSpan* in a diversity of artificial and real situations. This discussion clarifies the conditions that lead to a better performance of each algorithm. Since *SPaRSe* is an optimization of GSP, every constraint used by GSP can be applied without any change. Additionally, the use of regular expressions only requires the changes proposed in SPIRIT [4]. This makes SPaRSe and other candidate generation based methods competitive in conditions where restrictions are important.

References

[1] Agrawal R and Srikant R, “Mining Sequential Patterns”, in Int'l. Conf. Data Engineering (ICDE 95), (1995) 3-14

[2] Antunes C and Oliveira A.L: "Generalization of Pattern-Growth Methods for Sequential Pattern Mining with Gap Constraints" in *Int'l Conf Machine Learning and Data Mining*, (2003) 239-251

[3] Ayres J, Gehrke J, Yu T and Flannick J: "Sequential PAttern Mining using a Bitmap Representation" in *Int'l Conf Knowledge Discovery and Data Mining*, (2002) 429-435

[4] Garofalakis M, Rastogi R and Shim k, “Mining Sequential Patterns with Regular Expression Constraints”, in *IEEE Transactions on Knowledge and Data Engineering*, (2002), vol. 14, nr. 3, pp. 530-552

[5] Pei J, Han J. et al: “PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth” in *Int'l Conf Data Engineering*, (2001) 215-226

[6] Pei J and Han J: "Constrained frequent pattern mining: a pattern-growth view" in *SIGKDD Explorations*, (2002) vol. 4, nr. 1, pp. 31-39

[7] Srikant R, Agrawal R: “Mining Sequential Patterns: Generalizations and Performance Improvements”, in *Int'l Conf Extending Database Technology*. Springer (1996) 3-17

[8] Zaki M, "Efficient Enumeration of Frequent Sequences", in *ACM Conf. on Information Knowledge Management*, (1998) 68-75