Generalization of Pattern-growth Methods for Sequential Pattern Mining with Gap Constraints

Cláudia Antunes and Arlindo L. Oliveira

Department of Information Systems and Computer Science
Instituto Superior Técnico / INESC-ID
R. Alves Redol 9
1000 Lisboa, Portugal
claudia.antunes@dei.ist.utl.pt
aml@inesc-id.pt

Abstract. The problem of sequential pattern mining is one of the several that has deserved particular attention on the general area of data mining. Despite the important developments in the last years, the best algorithm in the area (PrefixSpan) does not deal with gap constraints and consequently doesn't allow for the introduction of background knowledge into the process. In this paper we present the generalization of the PrefixSpan algorithm to deal with gap constraints, using a new method to generate projected databases. Studies on performance and scalability were conducted in synthetic and real-life datasets, and the respective results are presented.

1 Introduction

With the rapid increase of stored data in digital form, the interest in the discovery of hidden information has exploded in the last decade. One approximation to the problem of discovery of hidden information is based on finding frequent associations between elements in sets, also called basket analysis. One important special case arises when this approach is applied to the treatment of sequential data. The sequential nature of the problem is relevant when the data to be mined is naturally embedded in a one dimensional space, i.e., when one of the relevant pieces of information can be viewed as one ordered set of elements. This variable can be time or some other dimension, as is common in other areas, like bioinformatics. We define sequential pattern mining as the process of discovering all sub-sequences that appear frequently on a given sequence database and have minimum support threshold. One challenge resides in performing this search in and efficient way.

In this paper, we present a generalization of the PrefixSpan algorithm to deal with gap constraints. A gap constraint imposes a limit on the separation of two consecutive elements of an identified sequence. This type of constraints is critical for the applicability of these methods to a number of problems, especially those with long sequences and small alphabets. The method we propose is based on the introduction of a new method to generate projected databases that efficiently stores the subsequences of all occurrences of each frequent element.

The paper is organized as follows: section 2 exposes the sequential pattern mining problem and its main application areas. Section 3 formalizes the sequential pattern mining problem and describes the specific problems addressed. Section 4 analyzes existing algorithms, paying particular attention to their behavior when dealing with gap constraints. Section 5 represents the main contribution of this work and presents a generalization of the PrefixSpan algorithm to deal with gap constraints. Section 6 describes the experimental results obtained with artificial and real-life data. Finally, section 7 draws the most relevant conclusions and points out guidelines for future research.

2 Sequential Pattern Mining

The problem of sequential pattern mining has deserved particular attention inside the general area of data mining. 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, or amino-acid sequences for problems in bioinformatics.

One particularly important problem in the area of sequential pattern mining is the problem of discovering all subsequences that appear on a given sequence database and have minimum support threshold. The difficulty is in figuring out what sequences to try and then efficiently finding out which of those are frequent [7].

One of the obvious applications of these techniques is in modeling the behavior of some entity, along time. For instance, using a database with transactions performed by customers at any instant, it is desirable to predict what would be the customer's next transaction, based on his past transactions. This type of concerns is one of the main goals of temporal data mining. Examples of these tasks are easily found on a number of areas, like the prediction of financial time series, patients' health monitoring and marketing, to cite only a few. With the increase of stored data in several domains and with the advances in the data mining area, the range of sequential pattern mining applications has enlarged significantly. Today, in engineering problems and scientific research sequential data appears, for example, in data resulting from monitoring sensor networks or spatial missions [4]. In healthcare, despite this type of data being a reality for decades (for example in data originated by complex data acquisition systems like ECGs or EECs), more than ever, medical staff is interested in systems able to help on medical research and on patients monitoring [6]. In businesses and finance, applications on the analysis of product sales, client behaviors or inventory consumptions are essential for today's business planning ([1], [3]). A survey of applications and methods used in temporal data mining has been presented recently [2].

Another relevant application of sequential pattern mining is in bioinformatics, where different characteristics of proteins and other biologically significant structures are to be inferred from mapped DNA sequences. Some important applications in this domain are on molecular sequence analysis, protein structure prediction, gene mapping, modeling of biochemical pathways and drug design [8].

3 Problem Definition

Several algorithms have been proposed to deal with the problem of sequential pattern mining, but they don't always share the same set of assumptions, which makes it difficult to compare them. In order to compare the performance of the two most significant approaches proposed to date, we present the basic notions needed to clearly define the problem of sequential pattern mining.

Definition 1. A *sequence* is an ordered list of elements called *items*. A sequence is *maximal* if it is not contained in any other sequence.

The number of elements in a sequence *s* is called the *length* of the sequence and is denoted by |s|. A sequence with length *k* is called a *k-sequence*. The i^{th} element in the sequence is represented by s_i. The empty sequence is denoted by <>. The result of the concatenation of two sequences *x* and *y* is a new sequence *s* denoted by *s=xy*.

Definition 2. A sequence $a=<a_1a_2... a_n>$ *is contained in* another sequence $b=<b_1b_2... b_m>$, or *a* is a *subsequence* of *b*, if there exist integers $1\leq i_1<i_2<\ldots<i_n\leq m$ such that $a_1=bi_1$, $a_2=bi_2$, …, $a_n=bi_n$.

A subsequence *s'* of *s* is denoted by $s'\subseteq s$, and by $s'\subset s$ if *s'* is a proper subsequence of *s*, i.e. if *s'* is a subsequence of *s* but is not equal to *s*.

When considering the existence of gap constraints, such as the use of a sliding window or some time constraints (as proposed by Srikant [7]), the notion of subsequence suffers some changes. In general, we can view this relaxation as an approximation to the original measure.

Definition 3. A sequence $a=<a_1a_2... a_n>$ is a δ-*distance subsequence* of $b=<b_1b_2... b_m>$ if there exist integers $i_1<i_2<\ldots<i_n$ such that $a_1=bi_1$, $a_2=bi_2$, …, $a_n=bi_n$. and $i_k-i_{k-1}\leq\delta$. Sequence $a=<a_1a_2... a_n>$ is a *contiguous subsequence* of $b=<b_1b_2... b_m>$ if *a* is a 1-distance subsequence of *b*, i.e., the elements of *a* can be mapped to a contiguous segment of *b*.

Note that a contiguous subsequence is a particular case of δ-distance subsequence (δ=1) and is equivalent to the original notion of subsequence. A δ-distance subsequence s' of s is denoted by $s' \subseteq_{\delta} s$. A contiguous subsequence s' of s is denoted by $s'\angle s$.

Definition 4. Given a database *D* of sequences and a user-specified minimum support threshold σ, a sequence is said to be *frequent* if it is *contained in* at least σ sequences in the database. A *sequential pattern* is a maximal sequence that is frequent.

Given a database D of sequences and a user-specified minimum support threshold σ, the problem of *mining sequential patterns* is to find all of the sequential patterns.

Note that beside the database and the minimum support threshold, the user may supply the δ, i.e. the maximum gap allowed between two consecutive elements in a sequence.

4 Existing Algorithms

4.1 Apriori-based Methods

The first approach to sequential pattern mining was the AprioriAll algorithm [1]. This algorithm follows the candidate generation and test philosophy, and looks for all patterns without considering the existence of gap constraints. It considers a sequence frequent if all of its elements are present (in the given order), on a sufficient number of sequences in the database.

```
AprioriAll (DB, min_sup) {
   L1 = { frequent 1-sequences };
   int k=2;
   while (Lk-1 ≠ ∅) {
      Ck = candidateGeneration(Lk-1, k);
      Ck = candidatePruning(Ck, k);
      Lk = supportBasedPruning(Ck);
      k ← k+1
   }
   return Maximal Sequences in ∪k Lk
}
candidateGeneration (Lk-1, k){
   Ck = ∅;
   for each a∈Lk-1
      for each b∈Lk-1
         if (∀ n, 1≤n≤k-2: an=bn)
            Ck ← Ck ∪ {a1…ak-2ak-1bk-1,
                a1…ak-2bk-1ak-1}
   return Ck;
}
```

Fig. 1. AprioriAll algorithm and its candidate generation method

The candidate generation in this case works by joining two frequent k-1-sequences when their maximal prefixes are equal. Each pair of such sequences originates two k-candidates, as illustrated in figure 1.

The great advantage of AprioriAll resides on its iterative nature and the use of the anti-monotonicity property. Using the frequent k-1-sequences, it generates the k-candidates, thus reducing the number of sequences to be searched in the database in comparison with exhaustive search. It also performs an additional reduction on the

number of candidates, by removing all the candidates that have some non-frequent k-1-subsequences, as shown in figure 2.

These reductions on the number of candidates are possible since the support of a sequence obeys the anti-monotonic property, which says that a k-sequence can't be frequent unless all of its k-1-subsequences are frequent.

```
candidatePruning (L_{k-1}, C_k, k) {
   for each s∈C_k
      if (∃ s' ⊂ s ∧ |s'|=k-1 ∧ s'∉L_{k-1})
         C_k ← C_k \ {s}
   return C_k;
}
```

Fig. 2. Candidate pruning based on anti-monotonic property

Naturally, the most expensive task is the support-based pruning, since it counts the support of each candidate on the full database. AprioriAll achieves best performance when the minimum support threshold is high and there are few frequent different 1-sequences, 2-sequences and so on. This leads to maximal pruning and reduces the number of support counts.

However when gap constraints are used, the AprioriAll algorithm cannot be applied directly. To illustrate this limitation, consider for example the data in Table 1 and a minimum support threshold of 40%, which means, in this case, that a pattern has to occur at least twice in the database. Additionally, assume that the gap constraint is equal to 1, which means that only contiguous sequences are allowed.

Table 1. Database example

Database
fgfgfg
acjcde
ababa
achcde
noqrst

The first step of AprioriAll will find *a*, *c*, *d* and *e* as frequent 1-patterns and *ac*, *cd* and *de* as frequent 2-patterns. However, the process will finish without discovering *cde*, since there are no 3-candidates. This is due to limitations in the candidate-generation method, which isn't able to generate all candidates. In fact, while the candidate generation process is complete when there are no gap constraints, it becomes incomplete when gap constraints are imposed.

Although to our knowledge, this property has never been stated clearly, it eventually led to the definition of a new method for candidate generation that does not suffer from this limitation.

```
candidateGeneration (L_{k-1}, k) {
   C_k = ∅;
   for each a∈L_{k-1}
      for each b∈L_{k-1}
         if (∀1≤n≤k-2: a_{n+1}=b_n )
            C_k ← C_k ∪ {a_1…a_{k-1}b_{k-1}}
   return C_k;
}
```

Fig. 3. Candidate generation in GSP method

The GSP algorithm [7] is an evolution of AprioriAll, allowing for the incorporation of gap constraints. The key difference between these two methods resides on the candidate generation procedure. The GSP algorithm creates a new candidate whenever the prefix of a sequence is equal to the suffix of another one, as illustrated on figure 3.

```
candidatePruning (L_{k-1}, C_k, k, gap) {
   if (gap≠1)
      for each s∈C_k
         if (∃ s' ⊂ s ∧ |s'|=k-1 ∧ s'∉L_{k-1})
            C_k ← C_k \ {s}
   else
      for each s∈C_k
         if (∃ s' ∠ s ∧ |s'|=k-1 ∧ s'∉L_{k-1})
            C_k ← C_k \ {s}
   return C_k;
}
```

Fig. 4. Candidate pruning in GSP method

The changes in the generation method imply changes in the candidate pruning process. If gaps are not allowed only candidates with some non-frequent contiguous subsequence need to be pruned. When gaps are allowed a sequence is pruned if it contains a non-frequent subsequence. Figure 4 shows the pseudo-code for candidate pruning method.

4.2 Pattern-growth Methods

Pattern-growth methods are a more recent approach to deal with sequential data 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 (DB, min_sup) {
   return MaximalSequences in run(<>,0, DB)
}

run (α, length, DB) {
   f_list = createsFrequentItemList(DB);
   for each b∈f_list {
      α' ← αb;
      L← L ∪ α'
      L←L∪run(α',length+1,createProjectedDB
(α',DB))
   }
   return L
}

createProjectedDB (α, DB) {
   for each s∈DB
      if (α ⊆ s) {
         β←s.postfix (α, 1)
         α-projDB← α-projDB ∪ {β}
      }
   return α-projDB;
}
```

Fig. 5. PrefixSpan algorithm and the creation of projected databases method

PrefixSpan [5] is the most promising of the pattern-growth methods and is based on recursively constructing the patterns, as shown in figure 5. Its great advantage is the use of projected databases. An α-projected database is the set of subsequences in the database, that are suffixes of the sequences that have prefix α. In each step, the algorithm looks for the frequent sequences with prefix α, in the correspondent projected database. In this way the search space is reduced in each step, allowing for better performances in the presence of small support thresholds.

Again, the PrefixSpan algorithm performs perfectly without gap constraints but is not able to deal with these restrictions. To illustrate that limitation, consider again the data in Table 1 and the conditions used before (minimum support threshold of 40% and a gap constraint equal to 1). It will find *a*, *c*, *d* and *e*, which will constitute *f_list*. Then it will call recursively the main procedure with α=*a* and an α-projected database equal to {*cjcde*, *baba*, *chcde*}. Next it will recursively proceed with α=*ac* and an α-projected database equal to {*jcde*, *hcde*}, which finishes this branch. Similarly for element *c*: *run* is called with α=*c* and an α-projected database equal to {*jcde*, *hcde*}. Since there is no frequent element at distance 1, the search stops and *cde* is not discovered. This happens because the α-projected database only maintains the suffix after the first occurrence of the last element of α.

5 Generalized PrefixSpan

In this section we show how PrefixSpan can be generalized to handle gap constraints, an important issue since gap constraints are important in many domains of application and PrefixSpan is the most efficient algorithm known for sequential pattern mining, especially in the situation of low support thresholds.

The generalization we propose for PrefixSpan (*GenPrefixSpan*) in order to be able to deal with gap constraints, is based on the redefinition of the method used to construct the projected database. Instead of looking only for the first occurrence of the element, every element's occurrence is considered. For example, in the previous example, the creation of the *c*-projected database would give as result {*jcde*, *de*, *hcde*, *de*} instead of {*jcde*, *hcde*} as before.

It is important to note that, including all suffixes after the element's occurrence changes the database and may change the number of times that each pattern appears. For instance, for the same example the *a*-projected database would be {*cjcde*, *baba*, *ba*, *chcde*}. In order to deal with this issue, associating an *id* to each original sequence in the database and guaranteeing that each sequence counts at most once for the support of each element is enough to keep an accurate count on the number of appearances of a given sequence.

```
initProjDB (α, DB, gap) {
   for each s∈DB {
      i←1;
      repeat {
         i←s.nextOccurrence(α, i+1);
         β←s.postfix(α, i)
         α-projDB←α-projDB ∪ {β}
      } until i + gap > |s|
   }
   return α-projDB;
}
```

Fig. 6. The new method to create projected databases

Figure 6 illustrates the new method we propose to create projected databases. This new approach is only needed in the first recursion level, since after this isolated step the database will contain all of the sequences starting with each frequent element.

So the generalized PrefixSpan will consist of two main steps: the discovery and creation of each frequent element projected database and the usual recursion, as shown in figure 7.

```
GenPrefixSpan (DB, min_sup, gap) {
     f_list = createsFrequentItemList(DB);
     for each b∈f_list {
        L← L ∪ b
        L← L ∪ run (b, 1,
                    initProjDB (b,DB, gap))
     }
     return MaximalSequences in L
}
```

Fig. 7. Generalized PrefixSpan main method

Note that when there is no gap constraint, the creation of projected databases is similar to the correspondent procedure defined in original PrefixSpan, since it only generates the projection relative to the first occurrence of α. In this manner, the performance of GenPrefixSpan and PrefixSpan are similar in the absence of gap constraints.

6 Comparison

In this section we present a comparative study between apriori-based and pattern-growth approaches with and without the presence of gap constraints. In order to do that, we use the AprioriAll, GSP and PrefixSpan algorithms in the absence of gap constraints, and the GSP and GenPrefixSpan algorithms in the presence of these restrictions.

All experiments were performed on a Pentium II with 300 MHz and 256MB of RAM. The sequences were generated and maintained in main memory during the algorithms processing.

All algorithms were implemented using an object-oriented approach allowing for the sharing of the basic methods used by the different algorithms and making all speed comparisons meaningful.

6.1 Experiences with Artificially Generated Data

To perform this study, we used a synthetic data set generator, based on a Zipf distribution, similar to others used on similar studies ([1], [5]). As parameters, this data generator receives the number of sequences, the average length of each sequence, the number of distinct items (or sequence elements) and a Zipf parameter that governs the probability of each item occurrence in the data set. The length of each sequence is chosen from a Poisson distribution with mean equal to the input parameter correspondent to the average length of each sequence (10 was the chosen value for the average sequence length).

The study is divided in two major sections: the scalability and the performance studies. Table 2 lists the parameters of the performed studies.

Table 2. Comparative studies performed

Study		DB size	Support	Gap	Alphabet Size
Scalability		—	33%	0	5
Performance	Variable Support	10000	—	0	5
	Variable Gap	10000	33%	—	5
	Variable Alphabet	10000	33%	0	—

Scalability Study. As it is shown in figure 8, the scalability of PrefixSpan when adapted to use gap constraints suffers some degradation, having a behavior similar to that of apriori-based algorithms. This leaves open the question of whether it is possible to generalize projection based methods, such as PrefixSpan, in a way that implies minimal impact, when compared with the situation where no gap constraints are used.

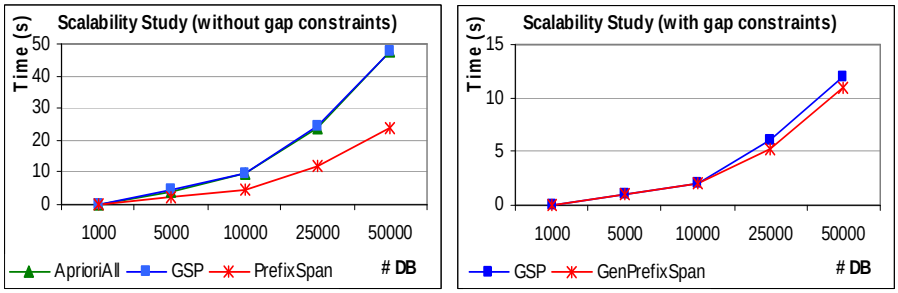

Fig. 8. Performance vs. database size

Note that the worst case to GenPrefixSpan (with gap constraints) is encountered when the database is composed of a significant number of sequences with the same element repeated several times. In this case, the projected database for each different element may be much bigger than the original database, violating the assumption that the size of projected database cannot exceed that of the original one, as is the case of the original version of PrefixSpan [5].

Performance with Variable Support. In terms of performance (figure 9), when the minimum support threshold varies, the behavior of PrefixSpan is similar with or without the use of gap constraints, with the same pattern of growth. This means that the advantages of PrefixSpan over apriori-based methods are still present in the situation of low value support thresholds.

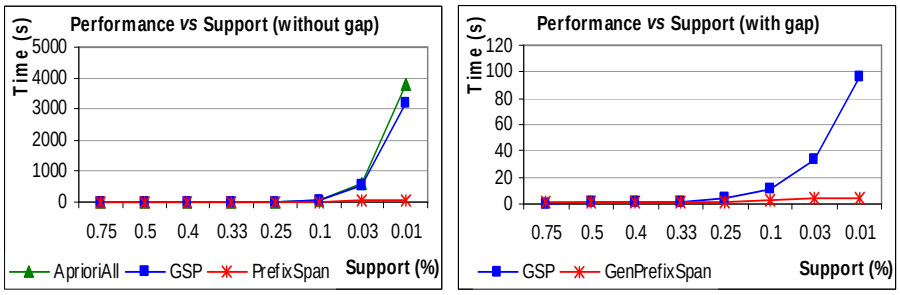

Fig. 9. Performance vs. minimum support threshold

Performance with Variable Gap Constraints

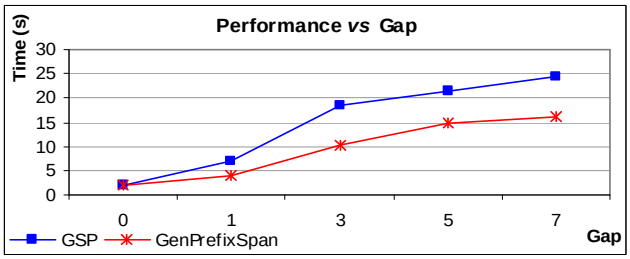

Fig. 10. Performance vs. gap value

As expected (fig. 10), when the gap constraint is relaxed, the performance decreases as in apriori-based algorithms, since the number of patterns to discover increases. Note that the difference between both methods increases with the relaxation of gap constraints. With this relaxation the probability to encounter frequent elements in the allowed gap is greater and consequently the search for the patterns is less time consuming.

Performance with Variable Alphabet Size.

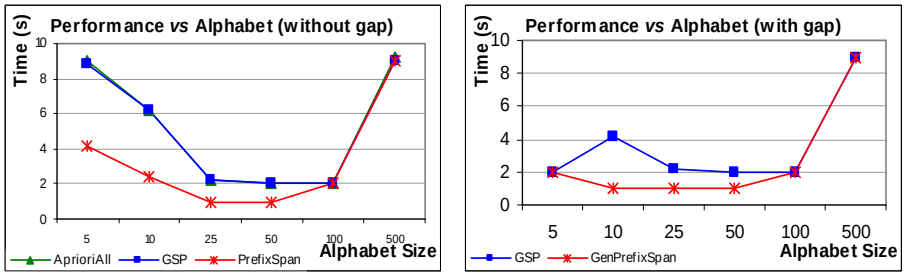

Fig. 11. Performance vs. alphabet size

The impact of the alphabet size on the performance of the methods did not lead to any clear conclusions. The PrefixSpan method seems to be better than apriori-based methods over all ranges of the alphabet size, but the results are inconclusive and the observed evolution is not easily explainable. It is worth noting that the change in alphabet size has a significant and non-trivial impact on the type and number of patterns present in the database, as is shown in figure 11.

6.2 Experiences with Real-World Data

To perform this study, we used the WWW server access logs from the web site of a discussion forum. The objective was to identify common patterns of access, in order to optimize the layout of the web site and, in the future, to identify and flag abnormal behaviors. The dataset is composed of about 7000 sequences, where each sequence represents the pages visited by one user when he enters the forum.

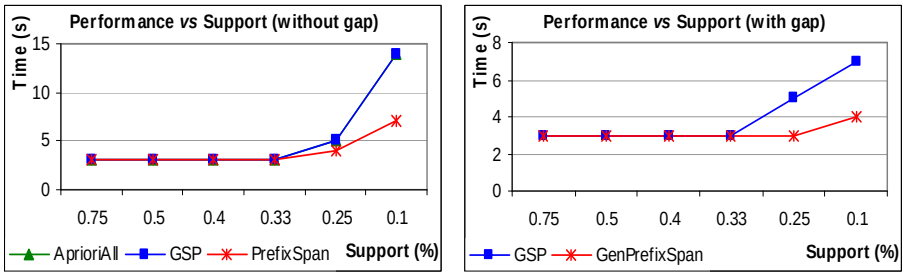

Fig. 12. Performance vs. minimum support threshold

In general, the results achieved with the real-life datasets (figure 12 and 13) confirm the results obtained with the synthetic dataset, despite the significantly different statistics of the problems. For this reason, we believe the results presented are relevant and applicable to a large range of actual problems.

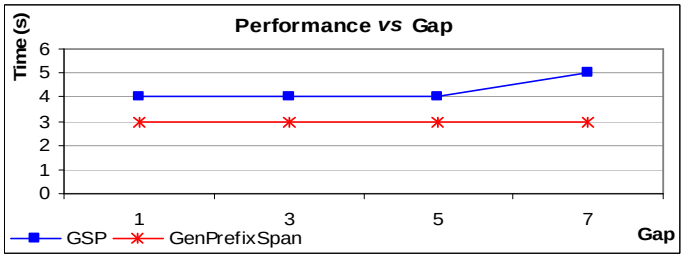

Fig. 13. Performance study with variable gap

7 Conclusions

In this paper we have presented the generalization of the PrefixSpan algorithm to deal with gap constraints. In order to achieve that goal, we have proposed a new method to generate projected databases that store the subsequences of all occurrences of each frequent element.

The modified PrefixSpan method keeps its performance advantages relatively to apriori-based algorithms in the more difficult situation of low support thresholds, although its relative advantage over these methods is reduced when compared with the high support thresholds situation.

The generalization of projection based methods to gap constrained sequential pattern problems is very important in many applications, since apriori-based methods are inapplicable in many problems where low support thresholds are used. In fact, the imposition of a gap restriction is critical for the applicability of these methods in areas like bioinformatics, which exhibit limited size alphabets and very long sequences. We are actively working in applying this methodology to the problem of motif finding in bioinformatics sequences, an area that can benefit very much from more sophisticated methods for sequential pattern analysis.

REFERENCES

1. Agrawal, R. and R. Srikant, "Mining sequential patterns", in *Proc. Int'l Conf. Data Engineering* (1995), 3-14
2. Antunes, C. and A. Oliveira, "Temporal data mining: an overview" in *Proc. Workshop on Temporal Data Mining* (KDD'01) (2001), 1-13
3. Fama, E., "Efficient Capital Markets: a review of theory and empirical work". *Journal of Finance* (1970) 383-417
4. Grossman, R. and C. Kamath *et all*, *Data Mining for Scientific and Engineering Applications*. Kluwer Academic Publishers (1998)
5. Pei, J, J. Han *et all* "PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth" in *Proc. Int'l Conf. Data Engineering* (ICDE 01) (2001)
6. Shahar, Y. and M.A. Musen, "Knowledge-Based Temporal Abstraction in Clinical Domains" in *Artificial Intelligence in Medicine* 8, (1996) 267-298
7. Srikant, R. and R. Agrawal, "Mining Sequential Patterns: Generalizations and Performance Improvements" in *Proc. Int'l Conf. Extending Database Technology* (1996) 3-17
8. Zaki, M., H.Toivonen and J. Wang, "Report on BIOKDD01: Workshop on Data Mining in Bioinformatics" in *SIGKDD Explorations*, vol. 3, nr. 2 (2001) 71-73