Inference of Sequential Association Rules Guided by Context-Free Grammars

Cláudia M. Antunes[1] and Arlindo L. Oliveira[2]

[1] Instituto Superior Técnico, Dep. Engenharia Informática, Av. Rovisco Pais 1, 1049-001 Lisboa, Portugal
claudia.antunes@dei.ist.utl.pt
[2] IST / INESC-ID, Dep. Engenharia Informática, Av. Rovisco Pais 1, 1049-001 Lisboa, Portugal
aml@inesc.pt

Abstract. One of the main unresolved problems in data mining is related with the treatment of data that is inherently sequential. Algorithms for the inference of association rules that manipulate sequential data have been proposed and used to some extent but are ineffective, in some cases, because too many candidate rules are extracted and filtering the relevant ones is difficult and inefficient. In this work, we present a method and algorithm for the inference of sequential association rules that uses context-free grammars to guide the discovery process, in order to filter, in an efficient and effective way, the associations discovered by the algorithm.

1 Introduction

With the rapid increase of stored data, the interest in the discovery of hidden information has exploded in the last decade. One important problem that arises during the discovery process is treating sequential data, and, in particular the inference of sequential association rules.

Association rules are a classical mechanism to model general implications of the form $X \Rightarrow Y$, and when applied to sequential patterns they model the order of events occurrence. Some of the main approaches to discover these sequential patterns are based on the well-known apriori algorithm [2], and they essentially differ on the data portions considered to generate pattern candidates [3], [11], [9]. However, the number of discovered rules is usually high, and the interest of most of them doesn't fulfill user expectations. Filtering them after the fact, i.e., after the generation phase, is inefficient and in some cases prohibitive.

In order to minimize this problem, an apriori-base algorithm, termed SPIRIT (Sequential Pattern mIning with Regular expressIons consTraints) [6] constrains the candidate generation with regular expressions. In this way, it is possible to focus the discovery process in accordance with the user expectations, and at the same time, to reduce the time needed to conclude the process. In this paper, we extend this approach to use Context Free Grammars as constraints.

The paper is organized as follows: section 2 describes some applications of sequential data mining. Section 3 presents the objectives of the paper and a summary of the principal apriori-based approaches to discover sequential association rules. Section 4 presents the adaptation of apriori to use context-free grammars as constraints to guide the discovery of frequent sequences. In section 5 an example of an interesting problem, which can be guided by a simple context-free grammar, is presented. Finally, section 6 presents the conclusions and points some possible directions for future research.

2 Applications of Sequential Data Mining

The ultimate goal of sequential pattern mining is to discover hidden and frequent patterns on sequences and sub-sequences of events. One important application is modeling the behavior of some entity. For instance, when using a database with transactions performed by customers at any instant, it is possible to predict what would be the customer's next transaction, based on his past transactions.

Temporal data mining is a particular case of the sequential pattern discovery problem. In the last two decades, the prediction of financial time series was one of the principal goals of temporal data mining [5]. However, with the increase of stored data in other domains and with the advances in the data mining area, the range of temporal data mining applications has enlarged significantly. Today, in engineering problems and scientific research temporal data appears, for example, in data resulting from monitoring sensor networks or spatial missions (see, for instance [7]). In healthcare, despite temporal data being a reality for decades (for example in data originated by complex data acquisition systems like ECG), more than ever medical staff is interested in systems able to help on medical research and on patients monitoring [13]. Finally, in finance, applications on the analysis of product sales, client behaviors or inventory consumptions are essential for today's business planning [3].

Another important 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 [15].

3 Sequential Pattern Mining

3.1 CFG Driven Inference of Temporal Association Rules

The aim of this paper is to present a method that uses context-free grammars as models that match interesting patterns hidden in the database, and uses these grammars to constrain the search. This is done by discovering sequential patterns with

an apriori-based algorithm whose search is constrained by the given context-free grammar. Changes in the grammar can then be applied in order to guide the search towards the desired objective, resulting in a methodology that effectively infers a context-free grammar that models interesting patterns in the database.

The first step on the process of inferring a grammar that supports frequent sequences is described, and consists on discovering the frequent sequences on the database that satisfies a given grammar. This problem is technically challenging and the main contribution of this paper is an efficient algorithm for this procedure.

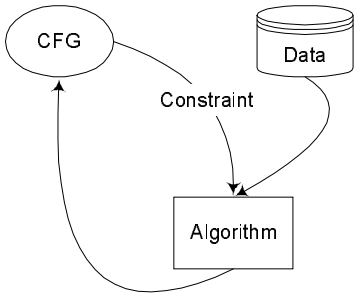

Fig. 1. Grammar inference process

The next step consists on evaluating the number, type and interest of the discovered sequences, and generalizing or specializing the given grammar, in order to adapt it to the obtained results.

Note that if the number of discovered sequences is too large, the grammar used to constrain the mining process is too generic and doesn't put enough restrictions on the discovery process. This means that the grammar should be restricted. On the other hand, if the number of discovered sequences is too small, it is necessary to generalize the grammar in order to accept a larger range of sequences.

Clearly, this is an highly non-automated method for grammatical inference, since the data mining practitioner herself is guiding the inference process. In practice, this may represent a very interesting method, since the use of association rule algorithms is usually encumbered by the difficulty in isolating the useful rules from the useless ones.

3.2 Sequential Association Rules

Sequential association rules are implications of the form $X \Rightarrow_T Y$ which states that if X occurs then Y will occur within T events [4], where X and Y are. Stating a rule in this new form, describes the impact of the occurrence of an event to the other event occurrence, within a specific number of events.

In general, the process of mining sequential association rules is composed mainly by the discovery of frequent patterns. Note that these patterns may be viewed as sequential association rules by themselves, since a sequence imposes an order on its

elements. One of the most common approaches to mining frequent patterns is the apriori method [2], which requires some adaptations for the case of sequential data.

Apriori acts iteratively generating the potential frequent k-itemsets (set of items) and pruning them by their support, until there are no candidates. A k-itemset is a set with k elements and is frequent when its support (the number of occurrences in the database) is greater than a user-specified threshold. In step k+1, the algorithm generates the new candidates by combining each two frequent k-itemsets and retaining the frequent ones to the next step.

This process could be used without loosing the property of completeness, because an itemset with length k isn't frequent unless all of its k-1 subsets are frequent. This property is known as anti-monotonicity [10], and allows for reducing significantly the number of candidates for which the support counting is done, reducing the time needed to discover the set of all frequent itemsets.

The AprioriAll [3] and its improvement GSP [14] are apriori adaptations for mining sequential patterns. The main difference to apriori is that candidates are sequences instead of itemsets, which means that the order in which each item appears is relevant. Another difference is that it is necessary to decide the value of T (the maximum gap between consecutive sequence elements), in order to restrict the impact of an element to another. Another important difference is on the notion of support: an entity supports a sequence/pattern if it is contained in the sequence that represents the entity, and there aren't more than T elements between two consecutive pattern elements. Stated in this form, an entity could only contribute once to increment the support of each pattern [3].

However, like apriori, the algorithm suffers from one main drawback: the lack of user-controlled focus.

SPIRIT Algorithms. SPIRIT is a family of apriori-based algorithms that uses a regular expression to constrain the mining process. The regular expression describes user expectations about data and preferences about the type of rules the user is interested on. In this way, it is used to constrain the candidates' generation. Given a database of sequences *D*, a user-specified minimum support threshold sup and a user-specified regular expression constraint *C*, the goal is to find all frequent and valid sequential patterns in *D*, in accordance with sup and *C* respectively [6].

The main algorithm is similar to AprioriAll, and is presented in figure 2. It consists essentially on three steps: candidate generation, candidate pruning and support pruning.

The candidate generation step, like in AprioriAll, is responsible for the construction of sequences with length k+1. However, the construction method depends on the chosen constraint. Since most of the interesting regular expressions are not anti-monotone, the construction consists on extending or combining k-sequences (that are frequent but not necessarily accepted by the regular expression) ensuring that all candidates satisfy the imposed constraint.

On the candidate pruning step, candidates which have some maximal k-subsequence valid and not frequent are removed, reducing the number of sequences passed to the next step and consequently, reducing the global processing time. Finally, on the support pruning step, the algorithm counts the support for each candidate and selects the frequent ones.

```
Procedure SPIRIT (D, C)
begin
   let C′:= a constraint weaker than C
   F:=F1:=frequent items in D that satisfy C′
   k:=2
   repeat{
    // candidate generation
    using C′ and F generate Ck:={potentially frequent
    k-sequences that satisfy C′}
    // candidate pruning
    let P:={s∈Ck: s has a subsequence t that satisfies
    C′ and t∉F}
    Ck:=Ck−P
    // support-based pruning
    scan D counting support for candidate k-sequences in
    Ck
    Fk:=frequent sequences in Ck
    F:=F∪Fk
    k:=k+1
   }until TerminatingCondition (F, C′) holds
   //enforce the original constraint C
output sequences in F that satisfy C
end
```

Fig. 2. SPIRIT algorithm [6]

With the introduction of constraints in the mining process, the algorithm restrains the search of frequent patterns in accordance to user expectations, and simultaneously it reduces the time needed to finish the mining process.

4 Using Context-Free Grammars as Constraints

Despite the fact that regular expressions provide a simple and natural way to specify sequential patterns, there are interesting patterns that these expressions are not powerful enough to describe.

Consider for example the following problem: a company wants to find out typical billing and payment patterns of its customers.

So, the specification language should be powerful enough to describe this constraint and exploit the fact that for well-behaved customers there are as many invoices as payments in any given order. If an invoice is represented by an *a* and a payment by a *b*, the language has to accept sequences like *abab* as well as *aabbab*, and rejects *aabbb* or *baab*. Note that, in the real world a payment transaction is always preceded by an invoice and not the other way around.

Note that no regular expression can be used to describe this problem since they are not able to record the number of occurrences for each element. However, context-free grammars are powerful enough to describe most of the structure in natural languages and simple enough to allow the construction of efficient parsers to analyze sentences [1]. For instance, the grammar $S \rightarrow aSbS|\varepsilon$ is able to model the problem stated above,

since the first expression imposes that if an invoice occurs, then a payment would also occurs in the future.

Context-free grammars have been widely used to represent programming languages and more recently, to model RNA sequences [12].

The SPIRIT algorithm exploits the equivalence of regular expressions to deterministic finite automata (DFA) to push the constraint deep inside the mining process. Using context-free grammars as constraints implies using non-deterministic push-down automata to do the same.

If we define a push-down automata as the tuple $(Q, \Sigma, \Gamma, \delta, q0, Z0, F)$ where Q is a finite set of states, Σ is an alphabet called the input alphabet, Γ is an alphabet called the stack alphabet, δ is a mapping from $Q\times(\Sigma\cup\{\varepsilon\})\times\Gamma$ to finite subsets of $Q\times\Gamma^*$, $q0\in Q$ is the initial state, $Z0\in\Gamma$ is a particular stack symbol called the start symbol and $F\subseteq Q$ is the set of final states[8]. The language accepted by a pushdown automaton is the set of all inputs for which some sequence of moves causes the pushdown automaton to empty its stack. The PDA equivalent to the grammar presented above would be

$$M = (\{q_1, q_2\}, \{a, b\}, \{S, X\}, \delta, q_1, S, \{q_2\})$$

with δ defined as follows:

$$\delta(q_1, a, S) = \{(q_2, push\ X)\},\ \delta(q_2, a, S) = \{(q_2, push\ X)\},$$
$$\delta(q_2, a, X) = \{(q_2, push\ X)\} \text{ and } \delta(q_2, b, X) = \{(q_2, pop)\}.$$

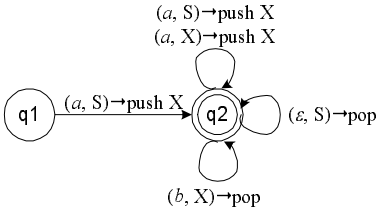

Fig. 3. A push-down automata equivalent to the grammar $S \rightarrow aSb|\varepsilon$.

This automaton is represented in figure 3, where, for example "(*a*, S)→*push* X" is used to indicate that when the input symbol is a and the top stack symbol is S, then X is pushed into the stack.

4.1 Using CFGs to Constrain the Search

Since a super-sequence of a given sequence s may belong to the CFG, even if s does not belong to the CFG, the anti-monotonicity property is not present. Therefore, using the PDA to restrict the candidate generation process is not straightforward. Four distinct relaxations to the original expression have been used with DFAs, namely:

- *Naïve*: an anti-monotonic relaxation of the constraint, which only prunes candidate sequences containing elements that do not belong to the language

alphabet. For example, if we consider the automaton defined in figure 1, only sequences with *a*'s and *b*'s are accepted by the *Naïve* constraint.

- *Legal*: the initial constraint is relaxed requiring that every candidate sequence is legal with respect to some state of the automaton equivalent to the constraint. A sequence is said to *be legal with respect to q* (with *q* a state of the automaton) if there is a path in the automaton, which begins in state *q* and is composed by the sequence elements. For example, if we consider the automaton like before, *a*, *ab*, *aab* or *b* are accepted as legal.
- *Valid suffix*: a constraint relaxation that only accepts candidate sequences valid with respect to any state of the automaton. A sequence is said to *be a valid suffix with respect to q* if it is legal with respect to *q* and achieves a terminal state. With the same automaton, *a* or *aa* aren't valid suffixes, but *b* or *ab* are.
- *Complete*: the constraint itself that imposes that every candidate sequence is accepted by the automaton. For example, *ab* or *aabb* are accepted.

A fifth possibility may be added to the above ones:

- *Valid prefix*: a reverse "valid suffix" relaxation, which requires that every candidate sequence is legal with respect to the initial state. Reversely, *a* or *ab* are valid prefixes but *b* is not.

The application of the naïve and complete alternatives with context-free grammars is straightforward. A sequence is accepted by the naive criterion in exactly the same conditions than before, and is accepted by the complete criterion if the sequence could be generated by the grammar that represents the constraint.

However, these two criteria are ineffective in many cases, for two different reasons. The naïve criterion prunes a small number of candidate sequences, which implies a limited focus on the desired patterns. The complete, can generate a very large number of candidates since the only way to apply it involves generating all strings s of a given length that belong to the language. The other two alternatives are, in many cases, significantly more effective in pruning the candidate list.

The extension of the legal and valid alternatives to context-free grammars is non trivial, since the presence of a stack makes it more difficult to identify when a given sequence is either legal or valid. However, it is possible to extend the notion of legality and validity of a sequence with respect to any state of the push-down automaton.

Consider the basic operations to manipulate stacks:

- λ.*push* X – introduces the element X on the top of the stack λ;
- λ.*top* – returns the element on the top of the stack λ;
- λ.*pop* – removes the element on the top of the stack λ;
- λ.*isEmpty* – verifies if the stack λ is empty.

Definition 1 A sequence s=<s1…sn> is legal with respect to state qi with stack λ, if and only if

- $|s|=1 \wedge \lambda.\text{isEmpty} \wedge \exists X \in \Gamma: \delta(q_i, s_1, X) \supset (q_j, op)$ with $op \in \{push, pop, no\ op\}$.
- $|s|=1 \wedge \exists X \in \Gamma \wedge \lambda.\text{top}=X: \delta(q_i, s_1, X) \supset (q_j, op)$ with $op \in \{push, pop, no\ op\}$.
- $\lambda.\text{isEmpty} \wedge \exists X \in \Gamma: \delta(q_i, s_1, X) \supset (q_j, pop) \wedge <s_2 \ldots s_n>$ is legal with respect to state q_j with stack λ.
- $\exists X \in \Gamma \wedge \lambda.\text{top}=X: \delta(q_i, s_1, X) \supset (q_j, op) \wedge <s_2 \ldots s_n>$ is legal with respect to state q_j with the resulting stack of applying the operation *op* to λ.

This means that any sequence with one element is legal with respect to a state, if it has a transition defined over the first element of the sequence. On the other cases, a sequence is legal with respect to a state if it has a transition defined over the first element of the sequence, and if the residual subsequence is legal with respect to the resulting state.

Consider again the automaton defined in figure 1:

- *a* is legal with respect to q_1 and the stack with S (rule i), since there is a transition from q_1 that could be applied $[\delta(q_1,a,S) \supset (q_2,\text{pushX})]$.
- *a* is also legal with respect to q_2 and the empty stack (rule ii) since there is also a transition from q_2 $[\delta(q_2,a,S) \supset (q_2,\text{pushX})]$.
- *b* is legal with respect to state q_2 and the empty stack since it has length-1 and there is a transition from q_2 with symbol *b* $[\delta(q_2,b,X) \supset (q_2,\text{pop})]$.
- *ab*, is naturally legal with respect to q_1 and the stack with S (rule iii), since from q_1 with *a*, q_2 is achieved and X is pushed into the stack. Then second symbol *b*, with X on the top of the stack the automaton performs another transition and a pop. Similarly *aba*, *abab*, *abab* and *aaba* are also legal with respect to q_1.
- *ba* is legal with respect to q_2 and the empty stack (rule iv). Since, with *b*, the automaton remains on q_2 and the stack empty, and with input *a*, X is pushed into the stack. Note that *ba* is a subsequence of *abab*, and consequently a sequence legal with respect to some state. Similarly, *bab*, *baab* and even *bbb* are legal with respect to q_2. Note that *bbb* is a subsequence of *aaabbb*, for example.

Note that pop is allowed on an empty stack, because it is possible that the sequence's first element doesn't correspond to a tr ansition from the initial state, or it may correspond to an element for which there are only transitions with pop operations, like the element b in the automaton in figure 1. If pop on the empty stack wasn't allowed, a simulation of every possible stack resulting from the initial to the current state would be necessary, in order to verify if the operation may be applied. This simulation could be prohibitive in terms of efficiency and would require a considerable effort.

Definition 2 A sequence s=<s1…sn> is said to be suffix-valid with respect to state qi with stack λ, if and only if:

- $|s|=1 \wedge \lambda.\text{isEmpty} \wedge \exists X \in \Gamma: \delta(q_i, s_1, X) \supset (q_j, op)$ with $op \in \{pop, no\ op\} \wedge q_j$ is a final state.
- $|s|=1 \wedge \exists X \in \Gamma \wedge \lambda.\text{top}=X: \delta(q_i, s_1, X) \supset (q_j, pop) \wedge q_j$ is a final state $\wedge$ ($\lambda.pop$).isEmpty.
- $\lambda.\text{isEmpty} \wedge \exists X \in \Gamma: \delta(q_i, s_1, X) \supset (q_j, pop) \wedge <s_2 \ldots s_n>$ is suffix-valid with respect to state q_j with stack λ.
- $\exists X \in \Gamma \wedge \lambda.\text{top}=X: \delta(q_i, s_1, X) \supset (q_j, op) \wedge <s_2 \ldots s_n>$ is suffix-valid with respect to state q_j, with the stack resulting of applying the operation *op* to λ.

This means that a sequence is suffix-valid with respect to a state if it is legal with respect to that state, achieves a final state and the resulting stack is empty.

Like before, consider the automaton defined in figure 1:

- *b* is a suffix-valid with respect to state q_2, since it is legal with respect to q_2, achieves a terminal state and the final stack is empty.
- *a* is not a suffix-valid, since any transition with a results on pushing an X into the stack, which implies a non empty stack.

Note that, in order to generate valid sequences with respect to any state, it is easier to begin from the final states. However, this kind of generating process is one of the more difficult when dealing with pushdown automata, since it is needed a simulation of their stacks.

In order to avoid this difficulty, using prefix validity instead of suffix validity could be an important improvement.

Definition 3 A sequence s=<s1…sn> is said to be prefix-valid if it is legal with respect to the initial state.

Sequences with valid prefixes are not difficult to generate, since the simulation of the stack begins with the initial stack: the stack containing only the stack start symbol.

Using the automaton defined in figure 1 again:

- *a* is a prefix-valid, since there is a transition with *a* from the initial state and the initial stack.
- *b* is not a prefix-valid, since there isn't any transition from the initial state with *b*.

The benefits from using the suffix-validity and prefix-validity are similar. Note that, like when using the suffix-validity, to generate the prefix-valid sequences with k elements, the frequent k-1-sequences are extended with the frequent 1-sequences, in accordance to the constraint.

Using these notions and an implementation of pushdown automata, it is possible to use context-free grammars as constraints to the mining process.

5 Example

This section exemplifies the application of the proposed algorithm to the discovery of sequential association rules, and simultaneously, to illustrate the process of inferring a grammar that supports the discovered rules. It begins with an example of the grammar inference process and finishes with the comparison of the results achieved with and without the use of context-free grammars to constrain the discovery of sequential rules.

Consider again the problem of identifying billing patterns of some company customers. A first definition of well-behaved customers may be: one customer is well-behaved if he doesn't receive an invoice before paying the previous one, and has done all his payments. This pattern could be specified by the regular grammar (a particular case of context-free grammars) $S \rightarrow abS \mid \varepsilon$.

Table 1. Data set used to exemplify the mining process

Data set
<*ababab*>
<*aabbab*>
<*aababb*>
<*babaab*>
<*abaabb*>

Supposing that the company has the dataset represented in table 1, the results of running our algorithm over the data set using the complete constraint are shown in table 2. Ck represents the set of candidate sequences with k elements and Fk the set of frequent k-sequences.

Table 2. Results achieved with the complete constraint defined by $S \rightarrow abS \mid \varepsilon$.

K	C_k	Support	F_k
1	-	-	-
2	<*ab*>	1.0	<*ab*>
3	-	-	-
4	<*abab*>	0.4	<*abab*>
5	-	-	-

Comparing the discovered sequences with the existent on the database, it is clear that the first approach to define well-behaved customers is too restrictive and the generalization of that notion is needed.

A simple generalization consists on considering that a well-behaved customer is a customer who always makes at least one payment after receiving one or two consecutive invoices and has made, at the end of the period, all its payments.

This constraint may be modeled by the grammar $S \rightarrow ASB|SCS|\varepsilon$ with $A \rightarrow aab$, $B \rightarrow b$ and $C \rightarrow ab$. The push-down automaton presented in figure 4 could be used to push the filtering imposed by this grammar inside the algorithm.

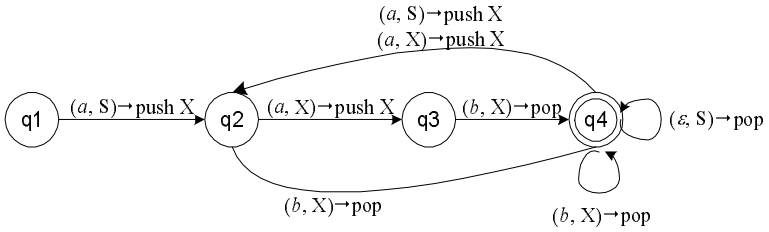

Fig. 4. Push-down automaton equivalent to grammar S→ASB|SCS|ε with A→aab, B→b and C→ab.

The results of running our algorithm over the data set using the new constraint are shown in table 3. We can suppose that, in this toy example, the results obtained are satisfactory in terms of the quality and number of the discovered sequences, and that no grammar generalization or specialization is required.

Table 3. Results achieved with the complete constraint defined by S→ASB|SCS|ε with A→aab, B→b and C→ab.

K	C_k	Support	F_k
1	-	-	-
2	<*ab*>	1.0	<*ab*>
3	-	-	-
4	<*aabb*> <*abab*>	0.4 0.4	<*aabb*> <*abab*>
5	-	-	-

Comparison to Apriori without Constraints

Table 4 presents the candidates generated by GSP and those generated by the algorithm adapted to use the complete context-free grammar constraint.

Table 4. Comparison of the results achieved with GSP and the complete constraint

	GSP			Complete		
K	C_k	Support	F_k	C_k	Support	F_k
1	<*a*> <*b*>	1.0 1.0	<*a*> <*b*>	–	–	–
2	<*aa*> <*ab*> <*ba*> <*bb*>	0.4 1.0 1.0 0.4	<*aa*> <*ab*> <*ba*> <*bb*>	<*ab*>	1.0	<*ab*>
3	<*aaa*> <*aab*> <*aba*>	0.0 0.8 0.8	<*aab*> <*aba*> <*abb*>	–	–	–

	GSP			Complete		
K	**C_k**	**Support**	**F_k**	**C_k**	**Support**	**F_k**
	<*abb*>	0.6	<*baa*>			
	<*baa*>	0.4	<*bab*>			
	<*bab*>	0.8				
	<*bba*>	0.2				
	<*bbb*>	0.0				
4	<*aaba*>	0.2				
	<*aabb*>	0.4	<*aabb*>			
	<*abaa*>	0.4	<*abaa*>			
	<*abab*>	0.4	<*abab*>	<*aabb*>	0.4	<*aabb*>
	<*baab*>	0.4	<*baab*>	<*abab*>	0.4	<*abab*>
	<*baba*>	0.4	<*baba*>			
	<*babb*>	0.0				
5	<*abaab*>	0.4				
	<*ababa*>	0.2				
	<*baabb*>	0.2	<*abaab*>	–	–	–
	<*babaa*>	0.2				
	<*babab*>	0.2				

Note that the number of candidates generated by the new algorithm is significantly less than with GSP. This difference appears because the constraint imposes that sequences are only accepted if they have an even number of elements. Note however, that when k=6 the GSP algorithm finishes, but the algorithm using the complete constraint continues counting the support for generated candidates.

Table 5. Comparison of the results achieved with the constraints "legal" and "prefix-valid"

	Legal			Prefix-valid		
K	**C_k**	**Support**	**F_k**	**C_k**	**Support**	**F_k**
1	<*a*>	1.0	<*a*>	<*a*>	1.0	<*a*>
	<*b*>	1.0	<*b*>			
2	<*aa*>	0.4	<*aa*>			
	<*ab*>	1.0	<*ab*>	<*aa*>	0.4	<*aa*>
	<*ba*>	1.0	<*ba*>	<*ab*>	1.0	<*ab*>
	<*bb*>	0.4	<*bb*>			
3	<*aab*>	0.8				
	<*aba*>	0.8	<*aab*>			
	<*abb*>	0.6	<*aba*>	<*aab*>	0.0	<*aab*>
	<*baa*>	0.4	<*abb*>	<*aba*>	0.8	<*aba*>
	<*bab*>	0.8	<*baa*>	<*abb*>	0.8	<*abb*>
	<*bba*>	0.2	<*bab*>			
	<*bbb*>	0.0				
4	<*aaba*>	0.2	<*aabb*>	<*aaba*>	0.2	
	<*aabb*>	0.4	<*abaa*>	<*aabb*>	0.4	<*aabb*>
	<*abaa*>	0.4	<*abab*>	<*abaa*>	0.4	<*abaa*>
	<*abab*>	0.4	<*baab*>	<*abab*>	0.4	<*abab*>

	Legal			Prefix-valid		
K	**C_k**	**Support**	**F_k**	**C_k**	**Support**	**F_k**
	<*baab*> <*baba*> <*babb*>	0.4 0.4 0.0	<*baba*>			
5	<*abaab*> <*ababa*> <*baabb*> <*babaa*> <*babab*>	0.4 0.2 0.2 0.2 0.2	<*abaab*>	<*abaab*> <*ababa*>	0.4 0.2	<*abaab*>

Note that on the results obtained with the legal constraint (presented in table 5), the difference between the numbers of generated candidates is less notorious. For example, C4 would only have seven elements versus the eight elements generated by GSP. However, using the valid-prefix constraint reduces the number of candidates significantly, focusing the discovered rules according the company expectations.

6 Conclusions

We presented a methodology and an algorithm that uses context-free grammars to specify restrictions to the process of mining temporal association rules using an Apriori-like algorithm.

Context-free grammars can model situations of interest to the data miner practitioner, and restrict significantly the number of rules generated. However, its application is not straightforward, since the restrictions imposed do not satisfy the anti-monotonicity property. We defined and proposed to use several different alternative restrictions that are a generalization of what has been proposed by other authors for regular grammars.

The results show that the additional expressive power of context free grammars can be used without incurring in any additional difficulties, when compared to the use of regular grammars, if the appropriate algorithms are used to restrict the search.

We have implemented these algorithms and tested them in small synthetic data sets with positive results. In the near future, we plan to use this approach in real data sets to empirically validate the efficacy and efficiency of the approach.

References

1. Allen, J.: Natural Languages Understanding, 2ndedition. The Benjamin/Cummings Publishing Company, Redwood City (1995)
2. Agrawal, R., Srikant, R.: Fast Algorithms for Mining Association Rules. In Proceedings of the International Conference on Very Large Databases (1994) 487-499
3. Agrawal, R., Srikant, R.: Mining sequential patterns. In Proceedings of the International Conference on Data Engineering (1995) 3-14
4. Das, G., Mannila, H., Smyth, P.: Rule Discovery from Time Series. In Proceedings of Knowledge Discovery in Databases (1998) 16-22

5. Fama, E.. Efficient Capital Markets: a review of theory and empirical work. Journal of Finance (1970) 383-417
6. Garofalakis, M., Rastogi, R., Shim, K.: SPIRIT: Sequential Pattern Mining with Regular Expression Constraint. In Proceedings of the International Conference on Very Large Databases (1999). 223-234
7. Grossman, R., Kamath, C., Kegelmeyer, P., Kumar, V., Namburu, R.: Data Mining for Scientific and Engineering Applications. Kluwer Academic Publishers (1998)
8. Hopcroft, J., Ullman, J.: Introduction to Automata Theory, Languages and Computation. Addison Wesley (1979).
9. Özden, B., Ramaswamy, S., Silberschatz, A.: Cyclic association rules. In Proceedings of the International Conference on Data Engineering (1998) 412-421
10. Ng, R., Lakshmanan, L., Han, J.: Exploratory Mining and Pruning Optimizations of Constrained Association Rules. In Proceedings of the International Conference on Management of Data (1998) 13-24
11. Ramaswamy, S., Mahajan, S., Silberschatz, A.: On the Discovery of Interesting Pa tterns in Association Rules. In Proceedings of the International Conference on Very Large Databases (1998) 368-379
12. Searls, D.B.: The Linguistics of DNA. American Scientist, 80 (1992) 579-591
13. Shahar, Y., Musen, M.A.: Knowledge-Based Temporal Abstraction in Clinical Domains. Artificial Intelligence in Medicine 8, (1996) 267-298
14. Srikant, R., Agrawal, R.: Mining Sequential Patterns: Generalizations and Performance Improvements. In Proceedings of the International Conference on Extending Database Technology (1996) 3-17
15. Zaki, M., Toivonen, H., Wang, J.: Report on BIOKDD01: Workshop on Data Mining in Bioinformatics. In SIGKDD Explorations, Volume 3, Issue 2 (2002) 71-73