Using Context-Free Grammars to Constrain Apriori-based Algorithms for Mining Temporal Association Rules

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. Algorithms for the inference of association with sequential information have been proposed and used but are ineffective, in some cases, because too many candidate rules are extracted. Filtering the relevant ones is usually difficult and inefficient. In this work, we present an algorithm for the inference of temporal association rules that uses context-free grammars to restrict the search process, in order to filter, in an efficient and effective way, the associations discovered by the algorithm. Moreover, we present experimental results that empirically evaluate its performance using synthetic data.

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 data with temporal dependencies, and, in particular, the discovery of temporal association rules.

When considering data with temporal information, the complexity of the mining process increases considerably, since the data related to each entity has to be viewed as a sequence of events. The ultimate goal of the process is to find behavior patterns among those sequences.

Association rules are a classical mechanism to model sequential patterns in general and temporal patterns in particular. Some of the main approaches to discover these patterns are based on the well-known *apriori* algorithm [2], and they essentially differ on the data portions considered to generate pattern candidates [3], [12], [10]. 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-based algorithm, termed SPIRIT (Sequential Pattern mIning with Regular expressIons consTraints) [7] 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. However, regular expressions are somehow limited, and cannot describe a number of interesting patterns. The aim of this paper is to present a generalization of this type of algorithms to use context-free grammars as constraints, in order to be able to represent those patterns. Such algorithm may be part of a broader process, able to identify and model sequential patterns, as described in [4].

The paper is organized as follows: section 2 presents a summary of the principal apriori-based approaches to discover sequential patterns. Section 3 presents the essential concepts related with context-free grammars and in section 4 an example of an interesting problem, which can't be described by any regular expression, is presented. Finally, section 5 presents the conclusions and points some possible directions for future research.

2 Temporal Data Mining

The ultimate goal of temporal data mining is to discover hidden and frequent patterns on sequences and sub-sequences of events. In this manner, it is possible to find patterns that model the behavior of an entity. For instance, 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.

In the last two decades, the prediction of financial time series was one of the principal goals of temporal data mining [6]. 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 been extended significantly. Today, in engineering problems and scientific research temporal data results, for example, from monitoring sensor networks or spatial missions (see, for instances [8]). 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 [14]. Finally in finance, applications on the analysis of product sales, client behaviors or inventory consumptions are essential for today's business planning [3].

2.1 Mining Temporal Association Rules

Temporal association rules are rules of the form $X \Rightarrow^{T} Y$ which states that if X occurs then Y will occur within time T [5], where X and Y are events and T an amount of time. Stating a rule in this new form, allows for controlling the impact of the occurrence of an event to the other event occurrence, within a specific time interval.

In general, the process of mining temporal association rules is composed mainly by the discovery of frequent patterns. Note that these patterns may be viewed as temporal 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.

The first thing to decide is the value of *T*, this is, the maximum gap between consecutive elements. The second issue is to define a new notion of *support*: an entity supports a sequence/pattern if the pattern is contained in the sequence that represents the entity, and there are no more than *T* elements between two consecutive elements. Stated in this form, an entity could only contribute once to increment the support of each pattern [3].

Like *apriori*, the algorithm *AprioriAll* [3] (and also its improvement *GSP* [15]) acts iteratively, generating the potential frequent *k*-sequences and verifying their support, until there are no candidates. In step *k*+1 the new candidates are generated by combining each two frequent *k*-patterns. This could be done because a sequence with length *k* isn't frequent unless all of its *k*-1 subsequences are frequent. This property is known as *anti-monotonicity* [11], and allows for reducing significantly the number of candidates, for which the support counting is done.

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 *R*, the goal is to find all frequent and valid sequential patterns in *D*, in accordance with *sup* and *R* respectively [7].

The main algorithm is similar to *AprioriAll*, and 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 aren't anti-monotone, the construction consists in 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 s*upport pruning* step, the algorithm counts the support for each candidate and selects the frequent ones.

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.

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. Note that, in the real world a payment transaction is always preceded by an invoice and not the other way around. In order to do 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*, but rejects *aabbb or baab*.

Note that no regular expression can be used to describe this problem [9], since DFA's are not able to record the number of occurrences for each element.

3 Context-Free Grammars

A *context-free grammar* (CFG) is a finite set of variables each of which represents a language. These languages are described recursively in terms of each other and primitive symbols called *terminals*. The rules relating the variables are called productions [9]. The term context-free comes from the feature that all productions must have a single symbol on its left-hand side, which means that the symbol could always be replaced by the right-hand side of the rule, no matter in what context it occurs.

This formalism is of great importance since it is powerful enough to describe most of the structure in computer languages and simple enough to allow the construction of efficient parsers to analyze sentences [1]. CFGs have been widely used to represent programming languages and more recently, to modeling RNA sequences [13].

Context-free grammars are strictly more powerful than regular expressions, since any language that can be generated using regular expressions can also be generated by a context-free grammar. On the other hand, there are languages that can be generated by context-free grammars that cannot be generated by any regular expression.

As an example, the grammar $S \rightarrow aSbS \mid \varepsilon$ 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.

3.1 Context-Free Grammars as Constraints

The SPIRIT algorithm exploits the equivalence of regular expressions to deterministic finite automata (DFA) to push the constraint deep inside the mining process. However, DFA aren't equivalent to context-free grammars, but with a simple memory an automaton could recognize languages generated by context-free grammars. Pushdown automata (PDA) are the counterpart machine for this type of grammars and are composed by an *input tape*, a *finite control* and a *stack* [9]. However, only non-deterministic pushdown automata are equivalent to context-free grammars.

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.

A pushdown automaton is defined by a tuple (Q, Σ, Γ, δ, q_0, Z_0, 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 $Q\times\Gamma^*$, $q_0\in Q$ is the initial state, $Z_0\in\Gamma$ is a particular stack symbol called the *start symbol* and $F\subseteq Q$ is the set of *final states* [9].

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:

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

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

This automaton is represented in figure 1,

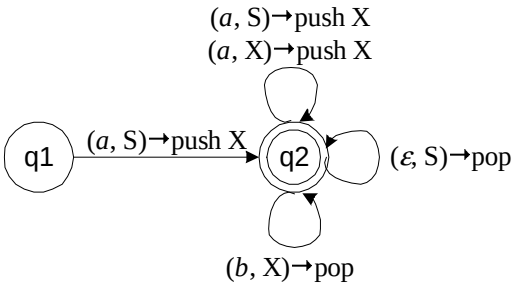

Figure 1 Pushdown automaton equivalent to the grammar S→aSb|ε.

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.

3.2 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. In the following definitions, consider that *push*, *pop*, *top* and *empty?* are the traditional operations to manipulate stacks.

Definition 3.1 A sequence $s=\langle s_1 \ldots s_n \rangle$ is *legal with respect to state* q_i with stack λ, if and only if

$|s|=1 \wedge \exists X \in \Gamma: \lambda.top=X \wedge \delta(q_i,s_1,X) \supset (q_j, op)$ with $op \in \{push, pop, no\ op\}$.

$|s|=1 \wedge \lambda.empty? \wedge \exists X \in \Gamma: \delta(q_i,s_1,X) \supset (q_j, op)$ with $op \in \{push, pop, no\ op\}$.

$\exists X \in \Gamma: \lambda.top=X \wedge \delta(q_i,s_1,X) \supset (q_j, op) \wedge \langle s_2 \ldots s_n \rangle$ is *legal with respect to state* q_j with stack $\lambda.op$.

$\lambda.empty? \wedge \exists X \in \Gamma: \delta(q_i,s_1,X) \supset (q_j, pop) \wedge \langle s_2 \ldots s_n \rangle$ is *legal with respect to state* q_j with stack 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 sequence's element. 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 [$\delta(q_1,a,S)\supset(q_2,pushX)$].
- *a* is also legal with respect to q_2 and the empty stack (rule ii) since there is also a transition from q_2 [$\delta(q_2,a,S)\supset(q_2,pushX)$].
- *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* [$\delta(q_2,b,X)\supset(q_2,pop)$].
- *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 transition 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 3.2 A sequence $s=\langle s_1 \ldots s_n\rangle$ is said to be a *suffix-valid with respect to state* q_i with stackλ, if and only if:

i) $|s|=1 \wedge \exists X\in\Gamma$: $\lambda.top=X \wedge \delta(q_i,s_1,X)\supset(q_j, pop) \wedge q_j$ is a final state $\wedge$ (λ.pop).*empty?*.
ii) $|s|=1 \wedge \lambda.empty? \wedge \exists X\in\Gamma$: $\delta(q_i,s_1,X)\supset(q_j, op)$ with $op\in$ {*pop, no op*} $\wedge q_j$ is a final state.
iii) $\exists X\in\Gamma \wedge \lambda.top=X$: $\delta(q_i, s_1, X)\supset(q_j, op)$ with $op\in$ {*pop, no op*} $\wedge\langle s_2 \ldots s_n\rangle$ is *suffix-valid with respect to* q_j, with stack $\lambda.op$.
iv) $\lambda.empty? \wedge \exists X\in\Gamma$: $\delta(q_i,s_1,X)\supset(q_j, pop) \wedge\langle s_2 \ldots s_n\rangle$ is *suffix-valid with respect to* q_j with stack 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 3.3 A sequence $s=<s_1 \ldots s_n>$ 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.

4 Experimental Results

In this section, we present a simple constraint to exemplify the results of using a context-free language as constraint and some experimental results when applying the described approaches on synthetic data sets.

The main goal of this study is showing that the performance and scalability of the algorithms when using context-free languages as constraints is not affected.

4.1 Simple Example

Consider again the problem of identifying billing patterns of some company customers. Suppose that the company considers 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 $S \rightarrow ASB \mid SCS \mid \varepsilon$ with $A \rightarrow aab$, $B \rightarrow b$ and $C \rightarrow ab$. The pushdown automaton presented in figure 2 could be used to push it inside the algorithm.

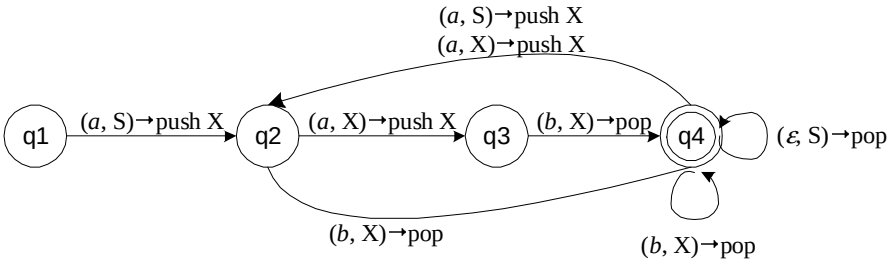

Figure 2 Pushdown automaton equivalent to the grammar S→ASB | SCS | ε with A→*aab*, B→*b* and C→*ab*.

Suppose that the company has the dataset represented in table 1, and it runs the different algorithms over the data set.

Table 1 Data set used to exemplify the mining process

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

Table 2 presents the candidates generated by `GSP` and those generated by the algorithm adapted to use the complete context-free grammar constraint (`SpiCFLComplete`).

Note that the number of candidates generated by the new algorithm is significantly less than with `GSP`. This enormous difference is achieved 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 2 Comparison of the results achieved with `GSP` and the complete constraint

	GSP				**SpiCFLComplete**	
K	**C_k**		**F_k**		**C_k**	**F_k**
1	<*a*>	<*b*>	<*a*>	<*b*>	-	-
2	<*aa*> <*ab*>	<*ba*> <*bb*>	<*aa*> <*ab*>	<*ba*> <*bb*>	<*ab*>	<*ab*>
3	<*aaa*> <*aab*> <*aba*> <*abb*>	<*baa*> <*bab*> <*bba*> <*bbb*>	<*aab*> <*aba*> <*abb*>	<*baa*> <*bab*>	–	–
4	<*aaba*> <*aabb*> <*abaa*> <*abab*>	<*baab*> <*baba*> <*babb*>	<*aabb*> <*abaa*> <*abab*>	<*baab*> <*baba*>	<*aabb*> <*abab*>	<*aabb*> <*abab*>
5	<*abaab*> <*ababa*> <*baabb*>	<*babaa*> <*babab*>	<*abaab*>		–	–

Note that with the legal constraint (`SpiCFLLegal` – which only accepts sequences legal with respect to some state) the difference between the numbers of generated candidates is less notorious. For example, C_4 would only have seven elements versus the eight elements generated by `GSP`.

Table 3 Comparison of the results achieved with the constraints "*legal*" and "*prefix-valid*"

	Legal				**PrefixValid**	
K	**C_k**		**F_k**		**C_k**	**F_k**
1	<*a*>	<*b*>	<*a*>	<*b*>	<*a*>	<*a*>
2	<*aa*> <*ab*>	<*ba*> <*bb*>	<*aa*> <*ab*>	<*ba*> <*bb*>	<*aa*> <*ab*>	<*aa*> <*ab*>
3	<*aab*> <*aba*> <*abb*>	<*baa*> <*bab*> <*bba*> <*bbb*>	<*aab*> <*aba*> <*abb*>	<*baa*> <*bab*>	<*aab*> <*aba*> <*abb*>	<*aab*> <*aba*> <*abb*>
4	<*aaba*> <*aabb*> <*abaa*> <*abab*>	<*baab*> <*baba*> <*babb*>	<*aabb*> <*abaa*> <*abab*>	<*baab*> <*baba*>	<*aaba*> <*aabb*> <*abaa*> <*abab*>	<*aabb*> <*abaa*> <*abab*>
5	<*abaab*> <*ababa*>	<*baabb*> <*babaa*> <*babab*>	<*abaab*>		<*abaab*> <*ababa*>	<*abaab*>

However, using the valid-prefix constraint (`SpiCFLPrefixValid`) reduces the number of candidates significantly, focusing the discovered rules according to the company expectations.

4.2 Performance and Scalability analysis

To perform this study, we used a synthetic data set generator similar to others used on similar studies [7]. 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 to govern 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.

All experiments were performed on a Pentium III with 731MHz and 384MB of RAM. The sequences were generated and maintained in main memory during the algorithms processing. This fact doesn't allow for tests with datasets with more than 100000 sequences.

All algorithms were implemented using an object-oriented approach and some basic methods are shared between several algorithms. For example, `SPIRIT(N)` and `SpiCFLNaive` are identical. The alphabet of each automaton was chosen to contain the most frequent elements in the dataset.

Relative Performance In order to compare the performance of the use and non-use of context-free languages, two experiments were done:

- *Use of constraints* – the performance of the proposed algorithms using context-free languages as constraints (`SpiCFL`) was compared to `GSP`. The context-free language used is represented as a push-down automata on figure 3, and is equivalent to the grammar $\mathbf{S \rightarrow aSb \mid bSa \mid a \mid b \mid \varepsilon}$. These experiments were performed with data sets with 25000 sequences, 5 different items, an average length size equal to 10 and the Zipf

parameter equal to 0.9. A maximum gap of 1 element is allowed for frequent sequences identification.

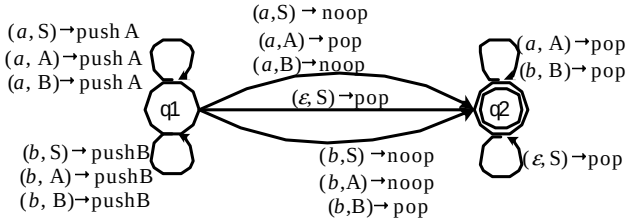

Figure 3 Pushdown automaton equivalent to the grammar *a*S*b* | *b*S*a* | *a* | *b* | ε.

Figure 4a) shows the execution time used for `GSP` and each variation of `SpiCFL`, when the minimum support decreases. Similarly, figure 4b) shows the evolution of the number of candidates for which the support is counted when the minimum support threshold decreases.

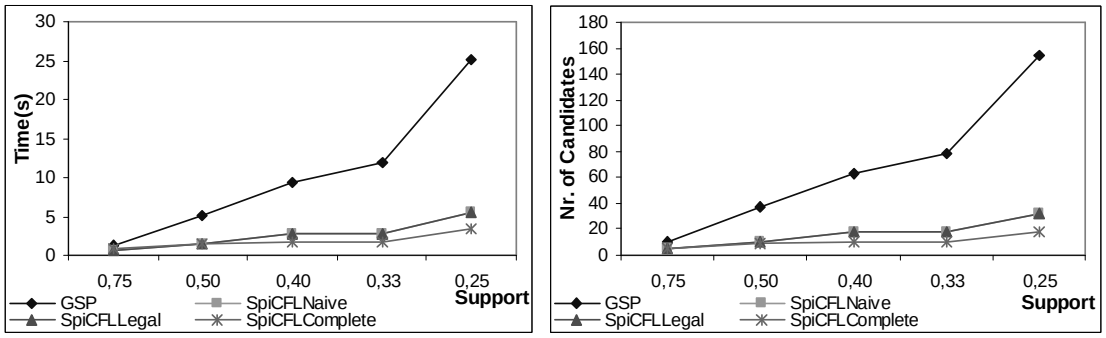

Figure 4 Comparison of `GSP` and `SpiCFL`: a) Execution Time *vs.* Minimum Support.
b) Number of Generated Candidates *vs.* Minimum Support.

As expected the execution time used by each algorithm is directly proportional to the number of candidates to which the support is counted. Like in `SPIRIT` the use of the complete constraint increases the performance significantly.

The most unexpected result is the similarity between the `Naïve` and `Legal` approach, which is justified by the weak restrictive power of the chosen language. Note that any sequence of *a*'s and *b*'s is *legal with respect to* q_1. For example, the language presented in the previous section has a stronger restrictive power than this one.

- *Use of particular cases of context-free languages* – one of the open questions is concerned with the way the performance is affected by the added complexity of push-down automata when compared to deterministic finite automata.

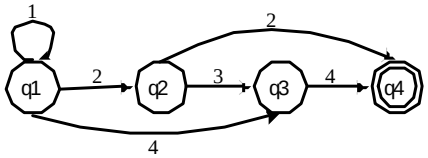

Figure 5 Deterministic finite automaton equivalent to regular expression 1* (22 | 234 | 44)

In order to clarify this question, we used a push-down automaton (represented in figure6) equivalent to a DFA (represented in figure5).

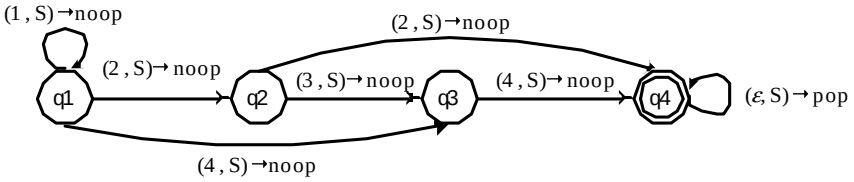

Figure 6 Push-down automaton equivalent to regular expression 1* (22 | 234 | 44):

Figure 7 shows that the execution time used by each pair of algorithms is identical and the number of candidates for which the support is counted is exactly the same for both algorithms.

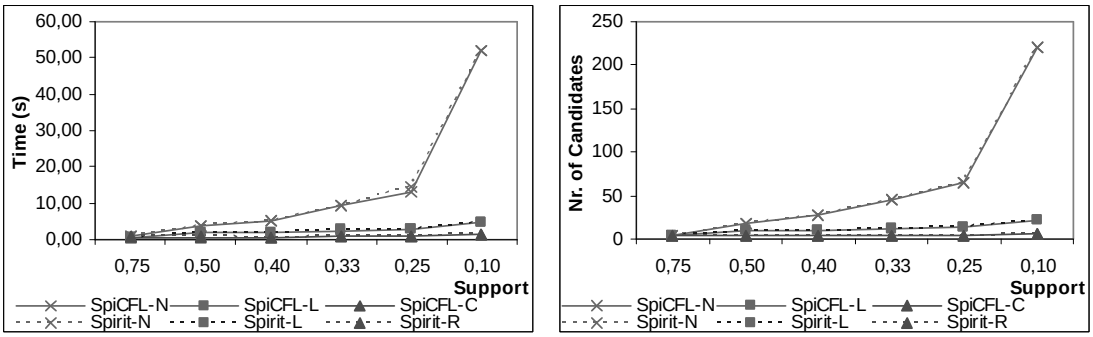

Figure 7 Comparison of `SpiCFL` and `SPIRIT`: a) Execution Time *vs.* Minimum Support.
b) Number of Generated Candidates *vs.* Minimum Support.

Scalability At last, the analysis of the performance of algorithms that filter the candidates using context-free languages shows that the execution time grows slower on these algorithms than on GSP (see figure8). This is explained by the fact that the number of candidates for which the support is counted, doesn't grow significantly, and consequently the execution time doesn't explode.

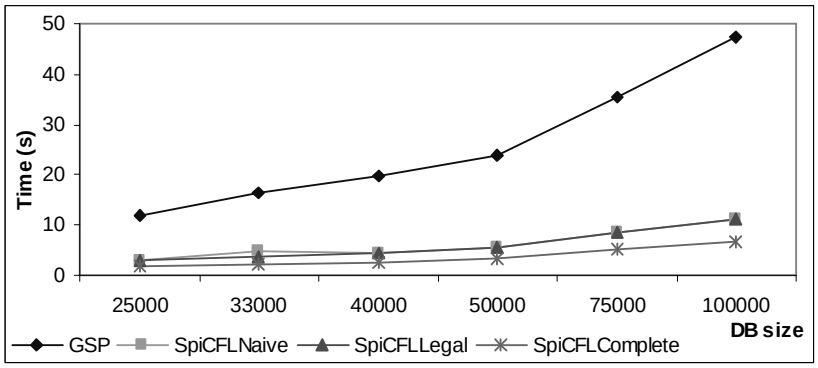

Figure 8 Comparison of `GSP` and `SpiCFL`: Execution Time *vs.* Dataset Size.

Since the support counting operation is one of the most time-consuming in the complete process of mining sequential patterns, decreasing the number of candidates also decreases the execution time, more than compensating by the increased processing.

5 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 when regular grammars are used.

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. 2nd edn. 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. Antunes, C.M., Oliveira, A.L.: Inference of Sequential Association Rules Guided by Context-Free Grammars. To appear in Proceedings of the International Conference on Grammatical Inference (2002).
5. Das, G., Mannila, H., Smyth, P.: Rule Discovery from Time Series. In Proceedings of Knowledge Discovery in Databases (1998) 16-22
6. Fama, E.. Efficient Capital Markets: a review of theory and empirical work. Journal of Finance (1970) 383-417
7. 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
8. Grossman, R., Kamath, C., Kegelmeyer, P., Kumar, V., Namburu, R.: Data Mining for Scientific and Engineering Applications. Kluwer Academic Publishers (1998)

9. Hopcroft, J., Ullman, J.: Introduction to Automata Theory, Languages and Computation. Addison Wesley (1979).

10. Özden, B., Ramaswamy, S., Silberschatz, A.: Cyclic association rules. In Proceedings of the International Conference on Data Engineering (1998) 412-421

11. 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

12. Ramaswamy, S., Mahajan, S., Silberschatz, A.: On the Discovery of Interesting Patterns in Association Rules. In Proceedings of the International Conference on Very Large Databases (1998) 368-379

13. Searls, D.B.: The Linguistics of DNA. American Scientist, 80 (1992) 579-591

14. Shahar, Y., Musen, MA.: Knowledge-Based Temporal Abstraction in Clinical Domains. Artificial Intelligence in Medicine 8, (1996) 267-298

15. Srikant, R., Agrawal, R.: Mining Sequential Patterns: Generalizations and Performance Improvements. In Proceedings of the International Conference on Extending Database Technology (1996) 3-17