Grammar Mining *

Siegfried Nijssen † Luc De Raedt ‡

Abstract

We introduce the problem of grammar mining, where patterns are context-free grammars, as a generalization of a large number of common pattern mining tasks, such as tree, sequence and itemset mining. The proposed system offers data miners the possibility to specify and explore pattern domains declaratively, in a way which is very similar to the declarative specification of regular expressions in popular scripting languages.

1 Introduction

In many application areas, such as bioinformatics and natural language processing, scripting languages that provide facilities for dealing with strings are very popular. Well-known examples of such scripting languages with good facilities for string processing and pattern matching are Perl, Python, Ruby and Tcl. One reason why these languages are popular, is that they provide general and easy ways to specify broad ranges of patterns. By using these systems users no longer have to study how to implement pattern matching algorithms, and can concentrate on the applications of interest.

The state of the art in pattern mining is different. For many pattern mining tasks in (semi-)structured data, such as substring, subsequence and subtree mining, separate systems have been proposed; a *general, declarative* approach to formulate a *domain* of patterns is missing. Our aim in this paper is to provide first steps to fill this gap. Our final goal is to provide users with a general *pattern mining system* in which it is sufficient to specify a pattern domain declaratively, after which the system is responsible for finding all patterns within this domain, without requiring the user to implement the search.

The approach that we propose in this paper is based on expressing patterns as grammars. Let us illustrate this for the pattern domain of *itemsets*. Assume given a binary matrix, represented by a set of bitstrings, as in the example below for a dabatase over items $\{1, 2, 3, 4\}$:

```
0110
1101
0111
```

Then the itemset $\{2, 3\}$ can be represented by the grammar $\{\mathtt{S} \rightarrow \mathtt{N11N}, \mathtt{N} \rightarrow \mathtt{0}|\mathtt{1}\}$, which parses only the first and third bitstring.

We propose a system which searches within a space of grammars as specified by a user, and by doing so, allows for rapid prototyping of pattern domains.

We preferred patterns expressed as grammars above patterns expressed as regular expressions, such as more common in scripting languages, for several reasons.

First, it is well-known that grammars can simplify the specification of patterns. Consider the following expression, which matches a string containing the letters `a`, `b` and `c` in that order:

$$(a|b|c|)^*a(a|b|c)^*b(a|b|c)^*c(a|b|c)^*$$

This expression is cumbersome to write down. It is more practical to introduce a variable $A = (a|b|c)^*$ and to rewrite the expression as $AaAbAcA$. However, we have now effectively obtained a grammar, in which we have a production rule for the nonterminal A. Languages such as Perl and Python are recognizing this, and are currently also gaining support to represent patterns beyond regular expressions.

Second, by using grammars we can express patterns which cannot be expressed in regular expressions. For instance, a popular pattern mining domain is that of *tree mining*. Trees can be represented as strings by using brackets [4]. To parse subtrees correctly, we need to match these brackets. Context-free grammars allow for such parsing, but regular expressions do not. We will show that by choosing grammars as patterns, our method is also applicable on tree mining tasks.

The main contributions of this paper are the following:

- we introduce the novel problem of grammar mining;
- we introduce the problem of user-guided grammar refinement;
- we propose fundamental grammar derivation rules;
- we propose algorithms for performing grammar mining.

*Supported by a Postdoc grant from the Research Foundation – Flanders and by EU FET IST project "Inductive Querying", contract number FP6-516169.

†Katholieke Universiteit Leuven, Belgium

‡Katholieke Universiteit Leuven, Belgium

The paper is organized as follows. In Section 2 we specify the problem that we are studying. In Section 3 we develop the principles of our grammar refinement strategies. We will present a simple language for specifying a space of grammars in Section 4, and we will show that this language is already general enough to deal with a wide range of pattern mining tasks. An algorithm for finding grammars is presented in Sections 5 and 6. Section 7 provides experiments; Section 9 concludes.

2 Problem Specification

We briefly review the key concepts concerning context-free grammars, cf. [21].

DEFINITION 1. *A* context-free grammar*, or* CFG*, G is specified by a quadruple (N, Σ, P, S), where*

> *N is the* nonterminal *or* variable *alphabet;*
> *Σ is the* terminal *alphabet; N and Σ are disjoint;*
> *$P \subseteq N \times (N \cup \Sigma)^*$ is a* finite *set of* productions*;*
> *S in N is the* sentence *or* start *symbol.*

A production is written as $A \rightarrow \alpha$ where A is in N and α is in $\alpha \in (N \cup \Sigma)^*$. This production is said to be an $A-$*production* for A, and α is the *right-hand side* of the production. We follow the convention that nonterminals are represented by uppercase letters, and that S is the start symbol. Terminals are represented by lowercase letters. Greek symbols represent arbitrary strings of terminals and nonterminals. A CFG *generates* a language by *rewriting*. If α can be derived from nonterminal A by rewriting using the productions P in G, we write that in G: $A \stackrel{*}{\rightarrow} \alpha$. More details on the derivation process will be provided below and can also be found in an introductory book on computer science, e.g., [21]. The language $L(G)$ parsed by the grammar contains all strings $s \in \Sigma^*$ for which $S \stackrel{*}{\rightarrow} s$. If $L(G_1) \subseteq L(G_2)$ grammar G_1 is more specific than G_2, denoted by $G_2 \succeq G_1$; G_2 is a generalization of grammar G_1. Grammars that are specializations of a grammar G, will also be referred to as being refinements of G.

The problem that we study in this paper can be formalized as follows:

DEFINITION 2. **Given**

- *a space of grammars $\mathcal{S}$*
- *a finite multiset of strings $\mathcal{D} \subseteq \Sigma^*$*
- *a support function $support(G, \mathcal{D})$ that computes the support of the grammar G in the data set $\mathcal{D}$*
- *a support threshold t*

Find: *all grammars $H \in \mathcal{S}$ such that $support(G, \mathcal{D}) \geq t$.*

To make our problem description complete we need to make the inputs of our problem more precise. In this section, we will introduce the *support* function. How to define a space of grammars is the topic of the next section.

The support function is similar to the support function that is used in many other frequent pattern miners. Essentially, it should count how many times a pattern matches the data; the function should have the property that if $G_2 \succeq G_1$ and $support(G_1, \mathcal{D}) \geq t$, it also holds that $support(G_2, \mathcal{D}) \geq t$. This property is called the anti-monotonicity property, and is of practical importance, as it allows us to ignore all refinements of a grammar for which $support(G_2, \mathcal{D}) < t$.

In our case, there are several possibilities for defining the support of a grammar:

- as the number of strings in $\mathcal{D}$ parsed by the grammar ('*entire transaction based support*');
- as the number of strings in $\mathcal{D}$ of which a substring is parsed by the grammar ('*substring transaction based support*');
- as the number of positions in the strings of $\mathcal{D}$ at which a substring starts that is parsed by the grammar ('*position based*').

Each of these functions is anti-monotonic. Please note that, strictly speaking, the second definition is redundant, as we can extend the grammars to parse remaining characters in the strings.

The first two support measures are useful in problems such as itemset mining, where we can distinguish clear separate examples or transactions. The third support measure is useful in databases that only contain one long string, such as query log databases.

Our methods can be extended to other types of constraints, such as correlation constraints, but we restrict ourselves to support only for reasons of simplicity.

3 Search Spaces

As pointed out in the previous section we need a mechanism for defining a search space. The approach that we propose in this paper is as follows. We assume given a general starting grammar $G = (N, \Sigma, P, S)$. We wish to develop an algorithm that finds grammars H such that $G \succeq H$. How to find such grammars?

Unfortunately, it is known that the $G \succeq H$ relation is undecidable [21]. Hence, we cannot hope to derive *complete* derivation rules that allow to find all grammars $G \succeq H$ for arbitrary G. At best, we can derive

sound derivation rules, i.e., derivation rules such that if $G_1 \vdash G_2$ it follows that $G_1 \succeq G_2$, although the other way around does not necessarily hold. These sound derivation rules should closely correspond to the rules needed in common pattern mining tasks.

Hence, we have the following procedural definition of the search space.

DEFINITION 3. *Given a set of* sound *derivation rules Q, and a starting grammar G. The search space $\mathcal{S}$ consists of all grammars H such that $G \vdash_Q H$, where $G \vdash_Q H$ expresses that grammar H can be derived from grammar G by the derivation rules in Q. We say that grammar H is a* refinement *of grammar G.*

In this section, we will review a number of sound derivation rules. Some of these rules are theoretically motivated, while others are motivated by applications. We will show how these rules relate to each other.

The derivation rules are illustrated on the following example grammar:

$$P_1 = \{\mathtt{S} \to \mathtt{AA}, \mathtt{A} \to \mathtt{aA}|\mathtt{bA}|\mathtt{a}|\mathtt{b}\} \tag{3.1}$$

From a theoretical perspective, the following two derivation rules are desirable.

Delete (D): Given a grammar with productions P, we delete a production $p \in P$.
Example refinement: We delete $\mathtt{A} \to \mathtt{aA}$ and $\mathtt{A} \to \mathtt{bA}$ to obtain
$$\{\mathtt{S} \to \mathtt{AA}, \mathtt{A} \to \mathtt{a}|\mathtt{b}\}$$

Copy (C): Given a grammar with productions P, we take a nonterminal in this grammar, and copy all productions that contain this nonterminal, replacing the old nonterminal with the new nonterminal in all possible ways.
Example refinement: We copy for A to obtain
$$\{\mathtt{S} \to \mathtt{AA}|\mathtt{A'A}|\mathtt{AA'}|\mathtt{A'A'},\ \mathtt{A} \to \mathtt{aA}|\mathtt{aA'}|\mathtt{bA}|\mathtt{bA'}|\mathtt{a}|\mathtt{b}, \mathtt{A'} \to \mathtt{aA}|\mathtt{aA'}|\mathtt{bA}|\mathtt{bA'}|\mathtt{a}|\mathtt{b}\}$$

DEFINITION 4. *We say that grammar $G_1 = (N_1, \Sigma, P_1, S)$ can be transformed into grammar $G_2 = (N_2, \Sigma, P_2, S)$, notation $G_1 \vdash_T G_2$, if we can derive G_2 from G_1 by applying the copy and delete rules (hence, $T = \{C, D\}$).*

For these two rules we can prove a weak completeness claim which states that every context-free language is a refinement of a grammar for the most general language Σ^*.

DEFINITION 5. *A grammar $G = (N, \Sigma, P, S)$ is in CNF (Chomsky Normal Form) iff every production is of one of these two forms:*

- *$A_1 \to A_2 A_3$, where $A_1, A_2, A_3 \in N$;*
- *$A \to \sigma$, where $A \in N$ and $\sigma \in \Sigma$.*

We say that two grammars G_1 and G_2 are syntactic variants if there is a renaming of the non-terminals which makes one grammar equal to the other.

THEOREM 3.1. *Given grammar $G_1 = (\{S\}, \Sigma, P, S)$ with $P = \{S \to SS\} \cup \bigcup_{\sigma \in \Sigma}\{S \to \sigma\}$ and grammar G_2, both in CNF. Then $G_1 \succeq G_2$ iff $G_1 \vdash_T G_3$ and G_3 is a syntactic variant of G_2.*

Proof. By n times copying for the initial nonterminal, we can generate the grammar with productions

$$P' = \{S_i \to S_j S_k, S_i \to \sigma | 1 \leq i, j, k \leq n, \sigma \in \Sigma\}.$$

After renaming the nonterminals in G_2 to nonterminals S_i, the set of productions in the resulting grammar G_3 must be a subset of P'. □

This claim is very general as it is known that every context-free language can be described by a grammar in CNF.

Despite its power, the copy operator is problematic from a practical point of view. We can apply it an unlimited number of times without deriving a grammar which is more specific; without limiting its use, we may get stuck in infinite recursions.

Possible other derivation rules, some of which do not have this disadvantage, are:

Select (S): Given a grammar with productions P and a nonterminal A, delete all productions for A, except one.
Example refinement: We delete all productions for A, except $\mathtt{A} \to \mathtt{a}$, to obtain
$$\{\mathtt{S} \to \mathtt{AA}, \mathtt{A} \to \mathtt{a}\}$$

Rewrite (R): Given a grammar with productions P, we apply one rewrite on one of the productions in P, and add the rewritten production.
Example refinement: We rewrite the underlined nonterminal in $\mathtt{S} \to \underline{\mathtt{A}}\mathtt{A}$ by using the production $\mathtt{A} \to \mathtt{aA}$,
$$\{\mathtt{S} \to \mathtt{AA}, \mathtt{S} \to \mathtt{aAA}, \mathtt{A} \to \mathtt{aA}|\mathtt{bA}|\mathtt{a}|\mathtt{b}\}$$

Substituting rewrite (SR): Given a grammar with productions P, we apply one rewriting operation on one of the productions in P, and remove the original production.
Example refinement: We rewrite the underlined nonterminal in $\mathtt{S} \to \underline{\mathtt{A}}\mathtt{A}$ by using the production $\mathtt{A} \to \mathtt{aA}$,
$$\{\mathtt{S} \to \mathtt{aAA}, \mathtt{A} \to \mathtt{aA}|\mathtt{bA}|\mathtt{a}|\mathtt{b}\}$$

Local copy (LC): Given a grammar with productions P, we take a nonterminal in the right-hand side of one production, replace this nonterminal with a new nonterminal, and copy all productions for this nonterminal, replacing the lefthand side of these productions with the new nonterminal.
Example refinement: We copy for the underlined nonterminal in $S \rightarrow \underline{A}A$,

$$\{\mathtt{S} \rightarrow \mathtt{AA}, \mathtt{S} \rightarrow \mathtt{A'A}, \mathtt{A'} \rightarrow \mathtt{aA}|\mathtt{bA}|\mathtt{a}|\mathtt{b}, \mathtt{A} \rightarrow \mathtt{aA}|\mathtt{bA}|\mathtt{a}|\mathtt{b}\}$$

Substituting local copy (SLC): Given a grammar with productions P, we apply a local copy on one nonterminal in one production, and delete that production.
Example refinement: We copy for the underlined nonterminal in $S \rightarrow \underline{A}A$,

$$\{\mathtt{S} \rightarrow \mathtt{A'A}, \mathtt{A'} \rightarrow \mathtt{aA}|\mathtt{bA}|\mathtt{a}|\mathtt{b}, \mathtt{A} \rightarrow \mathtt{aA}|\mathtt{bA}|\mathtt{a}|\mathtt{b}\}$$

The benefits of these additional derivation rules are:

- most of these operations effectively refine a grammar (except for the rewrite and local copy);
- they can allow to make larger, more constrained steps in the search space: for instance, by using the select operator instead of the delete operator, we can remove large numbers of productions in smaller numbers of steps, in applications where this is useful;
- they correspond more closely to operations on grammars that many computer scientists are familiar with; for instance, the rewrite operation is very similar to *resolution* in logic [12].

If we allow for deletes and rewrites only, we would obtain the following generality relation.

DEFINITION 6. *We say that grammar* $G_1 = (N_1, \Sigma, P_1, S)$ *is rewritten to grammar* $G_2 = (N_2, \Sigma, P_2, S)$, *notation* $G_1 \vdash_W G_2$, *if we can derive* G_2 *from* G_1 *by applying the rewrite and delete rules (hence,* $W = \{R, D\}$*).*

For many pattern domains these two derivation rules are sufficient. For instance, through (substituting) rewrites of

$$\{S \rightarrow L, L \rightarrow aL|bL|\lambda\}$$

we can create $\{S \rightarrow aL, L \rightarrow aL|bL|\lambda\}$, $\{S \rightarrow bL, L \rightarrow aL|bL|\lambda\}$, $\{S \rightarrow abL, L \rightarrow aL|bL|\lambda\}$, ...: all languages that first parse a string in $\{a, b\}^*$. Effectively, we could find all frequent substrings. On the other hand, not all of the rewrites allowed by $\vdash_W$ are necessary if we are interested in finding frequent subsequences; for instance, a rewrite of the rule $L \rightarrow aL$ is not necessary. We will discuss in the next section how we can more closely limit the refinement process to necessary refinements, as defined by the user.

We conclude this section with a discussion of how the derivation rules relate to each other. First, we can observe that the expressivity of rewrites and deletes is limited compared to the general copy and delete operations. Consider these two sets of productions:

$$P_1 = \{\mathtt{S} \rightarrow \mathtt{AA}, \mathtt{A} \rightarrow \mathtt{aA}|\mathtt{bA}|\mathtt{a}|\mathtt{b}\}$$

and

$$P_2 = \{\mathtt{S} \rightarrow \mathtt{AA'}, \mathtt{A} \rightarrow \mathtt{aA}|\mathtt{a}, \mathtt{A'} \rightarrow \mathtt{bA'}|\mathtt{b}\}$$

It is clear that the language of P_2 is a specialization of the language of P_1. However, rewrites cannot derive P_2 from P_1, as rewrites never introduce new nonterminals.

An important observation is hence the following.

THEOREM 3.2. *If* $G_1 \vdash_Q G_2$ *with* $Q \subseteq \{D, C, S, R, SR, LC, SLC\}$, *there is a grammar* G_3 *such that* $G_1 \vdash_T G_3$ *and* $L(G_3) = L(G_2)$.

Proof. For most of the rules this observation is rather straightforwardly proved. We limit ourselves here to showing how to emulate a rewrite. A rewrite starts with a local copy for the nonterminal that has to be rewritten. To perform the local copy, we first execute a general copy for this nonterminal. The productions that should result from the local copy are a subset of all these newly introduced productions. We delete all superfluous productions to obtain our intended set of productions. On the resulting set of productions, we apply a selection, as illustrated below, to remove the productions that were not selected in the rewriting:

$$\{\mathtt{S} \rightarrow \mathtt{AA}, \mathtt{A} \rightarrow \mathtt{aA}|\mathtt{bA}|\mathtt{a}|\mathtt{b}\} \xRightarrow{\text{Local copy for } \mathtt{S} \rightarrow \underline{\mathtt{A}}\mathtt{A}}$$
$$\{\mathtt{S} \rightarrow \mathtt{AA}, \mathtt{S} \rightarrow \mathtt{A'A}, \mathtt{A'} \rightarrow \mathtt{aA}|\mathtt{bA}|\mathtt{a}|\mathtt{b}, \mathtt{A} \rightarrow \mathtt{aA}|\mathtt{bA}|\mathtt{a}|\mathtt{b}\} \xRightarrow{\text{Deletes}}$$
$$\{\mathtt{S} \rightarrow \mathtt{AA}, \mathtt{S} \rightarrow \mathtt{A'A}, \mathtt{A'} \rightarrow \mathtt{aA}, \mathtt{A} \rightarrow \mathtt{aA}|\mathtt{bA}|\mathtt{a}|\mathtt{b}\}$$

Obviously, a selection consists of a set of deletes. The latter grammar is equivalent to a grammar that contains $\mathtt{S} \rightarrow \mathtt{aAA}$ instead of $\{\mathtt{S} \rightarrow \mathtt{A'A}, \mathtt{A'} \rightarrow \mathtt{aA}\}$. □

The following corollary follows then.

COROLLARY 3.1. *Given two grammars such that* $G_1 \vdash_W G_2$. *Then there is a grammar* G_3 *with* $L(G_3) = L(G_2)$ *such that* $G_1 \vdash_T G_3$.

4 Language Bias

In the previous sections we introduced the general problem of grammar mining, and the possible derivation

rules that one can apply on grammars. We showed that the copy and delete operations are the most powerful, but we need mechanisms to limit their application to deal with concrete pattern domains. In this section, we will propose one possible language for specifying a *language bias*. The language bias defines which derivations should be performed during the search. This language should enable users to restrict the search such that effectively itemsets, trees, sequences and other pattern domains can be mined.

The language that we propose in this section allows users to specify the necessary applications of the delete (D), select (S) and substituting local copy (SLC) derivation rules. We will see that the proposed mechanism is sufficient to deal with a large number of pattern domains.

4.1 Specification Language. Our approach is to augment productions with special markers. These markers do not only indicate if removal of productions is allowed, but also specify which nonterminals are eligible for local copying, and specify what happens once a production is copied. A newly generated grammar also includes new markers to reflect what is allowed in the new grammar.

In the augmented grammars, every production is marked as

- deletable ($\ominus$);
- selectable ($\oplus$);
- fixed ($\otimes$).

Productions for a nonterminal may only be marked as selectable if there are at least two productions for that nonterminal and all the productions for that nonterminal are also marked that way.

Every nonterminal in the righthand side of a production can be marked as

- eligible for local copying ($\bullet$);
- eligible for local copying after copying ($\circ$).

An example of an augmented grammar is

$$\{\otimes \mathtt{S} \rightarrow \bullet \mathtt{AA}, \otimes \mathtt{A} \rightarrow \mathtt{a} \circ \mathtt{A} | \mathtt{b} \circ \mathtt{A} | \mathtt{a} | \mathtt{b}\}$$

In this case, all productions are marked as fixed.

The markers are used as follows. For nonterminals marked with a $\bullet$, and occurring in a *fixed* production, we perform substituting local copies. The copied productions are marked as selectable. The markers in these productions are also copied, where we update $\circ$ markers into $\bullet$ markers in the copies.

However, given that these copied markers do not occur in fixed productions, we do not need to recursively copy further; hence we avoid unlimited copying. Only after we have selected a selectable production does it become fixed, and is further copying possible.

In our example, the result is

$$\{\otimes \mathtt{S} \rightarrow \mathtt{A'A}, \oplus \mathtt{A'} \rightarrow \mathtt{a} \bullet \mathtt{A} | \mathtt{b} \bullet \mathtt{A} | \mathtt{a} | \mathtt{b}, \otimes \mathtt{A} \rightarrow \mathtt{a} \circ \mathtt{A} | \mathtt{b} \circ \mathtt{A} | \mathtt{a} | \mathtt{b}\}$$

In this grammar, we can only perform a selection; local copying is not yet possible as the nonterminals marked with $\bullet$ are not part of fixed productions. After a selection, the selected production becomes fixed, and we can perform a local copy again. A possible resulting grammar is

$$\{\otimes \mathtt{S} \rightarrow \mathtt{A'A}, \otimes \mathtt{A'} \rightarrow \mathtt{a} \bullet \mathtt{A}, \otimes \mathtt{A} \rightarrow \mathtt{a} \circ \mathtt{A} | \mathtt{b} \circ \mathtt{A} | \mathtt{a} | \mathtt{b}\}$$

In this grammar, we can perform a local copy again, and so on.

Effectively, this simple language allows us to specify for which nonterminals substituting rewrites are allowed ($\bullet$), and once such a rewrite is performed, whether further rewriting is allowed ($\circ$). The rewrites are split into two phases: a local copy phase, and a selection phase. The first phase does not specialize the language, only the second phase does. The rule that forbids local copies for productions that are not fixed, prevents that we can endlessly copy without obtaining more specific languages.

4.2 Applications. Let us now study how we can use this language to specify a large number of frequent pattern mining tasks. In all our formulations, we assume entire transaction based support.

Disjunctions. Assume given a set of items $\mathcal{I}$. We wish to find itemsets $I \subseteq \mathcal{I}$ such that there are many transactions $T \subseteq \mathcal{I}$ in a transactional database $\mathcal{D}$ such that $T \cap I \neq \emptyset$. Then the following bias allows us to find such disjunctions of items:

$$\{\ominus \mathtt{S} \rightarrow \mathtt{L}i\mathtt{L}, \otimes \mathtt{L} \rightarrow i\mathtt{L} | \lambda : i \in \mathcal{I}\},$$

Initially, this grammar parses any string that contains at least one symbol. After several deletions, the grammar only parses strings that contain at least one item of an increasingly smaller subset of items.

An example of a grammar wich can be found for $\mathcal{I} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ is the following, which represents the pattern $\mathtt{a} \vee \mathtt{b}$:

$$\{\mathtt{S} \rightarrow \mathtt{LaL} | \mathtt{LbL}, \mathtt{L} \rightarrow \mathtt{aL} | \mathtt{bL} | \mathtt{cL} | \lambda\}$$

Supersets. Assume given a set of items $\mathcal{I}$. We wish to find itemsets $I \subseteq \mathcal{I}$ such that there are many transactions $T \subseteq \mathcal{I}$ in a transactional database $\mathcal{D}$ such that $T \subseteq I$. Then the following bias allows us to find supersets of items:

$$\{\ominus \mathtt{S} \rightarrow i\mathtt{S}, \otimes \mathtt{S} \rightarrow \lambda : i \in \mathcal{I}\},$$

Initially, this grammar parses a string containing all symbols in $\mathcal{I}$. After several deletions, the grammar only parses strings that contain subsets of these symbols; effectively, every grammar denotes a superset of the itemsets that should be covered.

For example, the following grammar parses all strings containing items `a` and `b`:

$$\{\mathtt{S} \rightarrow \mathtt{aS}|\mathtt{bS}|\lambda\}.$$

Substrings. Assume given a set of items $\mathcal{I}$, the following bias allows us to find frequent substrings:

$$\{\otimes \mathtt{S} \rightarrow \mathtt{N} \bullet \mathtt{N}, \otimes \mathtt{N} \rightarrow i \circ \mathtt{N}|\lambda : i \in \mathcal{I}\}$$

Initially, this grammar parses any string containing the symbols $\mathcal{I}$. After several rewrites, the grammar will only parse strings that contain a particular string in $\mathcal{I}^*$, for example the grammar:

$$\{\mathtt{S} \rightarrow \mathtt{NN}', \mathtt{N}' \rightarrow \mathtt{aN}'', \mathtt{N}'' \rightarrow \mathtt{bN}, \mathtt{N} \rightarrow \mathtt{aN}|\mathtt{bN}|\mathtt{cN}|\lambda\},$$

which parses only strings containing the substring `ab`.

Subsequences. The problem of subsequence mining can be formulated as follows:

$$\{\otimes \mathtt{S} \rightarrow \bullet \mathtt{R}, \otimes \mathtt{R} \rightarrow \mathtt{N}i \circ \mathtt{R}|\mathtt{N}, \otimes \mathtt{N} \rightarrow i\mathtt{N}|\lambda : i \in \mathcal{I}\}$$

After several rewrites, the grammar will only parse strings that contain a string in $\mathcal{I}^*$. In contrast to the previous grammar, it is possible that we need to skip symbols in the data string to find this substring.

For instance, the subsequence `ab` is represented by:

$$\{\mathtt{S} \rightarrow \mathtt{R}_1, \mathtt{R}_1 \rightarrow \mathtt{NaR}_2, \mathtt{R}_2 \rightarrow \mathtt{NbR}_3, \mathtt{R}_3 \rightarrow \mathtt{N}, \mathtt{N} \rightarrow \mathtt{aN}|\mathtt{bN}|\mathtt{cN}|\lambda\}$$

Itemsets. There are two ways to formulate the frequent itemset mining problem. Assume that we sort the symbols in the data before starting the mining process, then the first option is to use the bias for finding subsequences to find sorted subsequences, which correspond to itemsets. The second option is to represent the data in a binary matrix with n columns, and use a set of nonterminals I_i for all $i \in \mathcal{I}$, as follows:

$$\{\otimes \mathtt{S} \rightarrow \mathtt{I}_1 \ldots \mathtt{I}_n, \otimes \mathtt{I}_i \rightarrow \mathtt{1}, \ominus \mathtt{I}_i \rightarrow \mathtt{0} : i \in \mathcal{I}\}$$

In this bias, by repeatedly dropping productions $\mathtt{I}_i \rightarrow \mathtt{0}$ we are effectively adding items in the itemset.

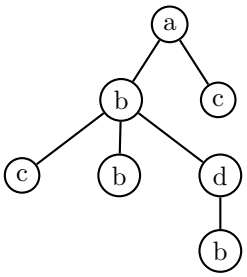

Figure 1: An example of an ordered tree

Ordered Induced Subtrees. Assume that we represent trees using strings such as

$$(a(b(c)(b)(d(b)))(c)),$$

which corresponds to the tree in Figure 1.

Then the following grammar parses such trees.

$$\{\oplus \mathtt{S} \rightarrow (i \bullet \mathtt{L}), \otimes \mathtt{L} \rightarrow \mathtt{L}(i \circ \mathtt{L}) \circ \mathtt{L}|\lambda : i \in \mathcal{I}\}$$

An example of a grammar that can be obtained after several rewrites is

$$\{\otimes \mathtt{S} \rightarrow (\mathtt{aL}_1), \otimes \mathtt{L}_1 \rightarrow \mathtt{L}(\mathtt{bL}_2) \bullet \mathtt{L}, \otimes \mathtt{L}_2 \rightarrow \mathtt{L}(\mathtt{c} \bullet \mathtt{L}) \bullet \mathtt{L}, \otimes \mathtt{L} \rightarrow \mathtt{L}(i \circ \mathtt{L}) \circ \mathtt{L}|\lambda : i \in \mathcal{I}\}$$

This grammar represents a (linear) tree with nodes labeled a, b and c. If we use a substring based support measure, the grammar which represents a tree can be matched also lower down the tree. Effectively, each grammar represents an induced subtree in the data [2].

Ordered Embedded Subtrees. We can use similar ideas to find embedded induced subtrees [25]. The difference between embedded subtrees and induced subtrees is as follows. For a tree to be an *induced* subtrees, parent-child relationships in the pattern have to be mapped to parent-child relationship in the data. In *embedded* subtrees the matching relation is less strict: parent-child relationships in patterns can also be mapped to ancestor-descendant relationships in the data.

We can represent a tree with a in the root and d as its only child (a tree which is an embedded subtree of the tree in Figure 1, but not an induced subtree), together with an embedding relation, as follows.

$$\{\mathtt{S} \rightarrow \mathtt{L}_1, \mathtt{L}_1 \rightarrow \mathtt{L}(i\mathtt{L}_1)\mathtt{L}|\mathtt{L}_2, \mathtt{L}_2 \rightarrow \mathtt{L}(\mathtt{a}\mathtt{L}_3)\mathtt{L},$$
$$\mathtt{L}_3 \rightarrow \mathtt{L}(i\mathtt{L}_3)\mathtt{L}|\mathtt{L}_4, \mathtt{L}_4 \rightarrow \mathtt{L}(\mathtt{d}\mathtt{L})\mathtt{L}, \mathtt{L} \rightarrow \mathtt{L}(i\mathtt{L})\mathtt{L}|\lambda : i \in \mathcal{I}\}$$

This grammar can be obtained through copies and deletes from this general grammar:

$$\{\mathtt{S} \rightarrow \mathtt{L}, \mathtt{L} \rightarrow \mathtt{L}(i\mathtt{L})\mathtt{L}|\lambda : i \in \mathcal{I}\}$$

However, the specification language that we presented in this section does not allow for the specification of this derivation, as in the specialized grammar, a copied nonterminal occurs both in the left-hand and righthand side of the same production ($\mathtt{L}_1 \rightarrow \mathtt{L}(i\mathtt{L}_1 L)$. To deal with such rewrites, we developed an extension of our bias specification language, in which we can also perform local copies for *sets of productions*. However, we skip the details of this extension here.

5 Algorithm

In this section we propose a depth-first algorithm for discovering all grammars that are frequent in a database of strings. An outline of this algorithm is given in Algorithm 1. The setup of this algorithm is similar to that of other frequent structure mining algorithms [22, 23, 10].

In this algorithm we distinguish two types of refinements, as directed by the user-defined bias:

- restrictions (usually deletions and selections), which specialize a language;
- extensions (usually local copies), which generate equivalent languages.

Each restriction is a set of productions that may be removed from the grammar G'. With $G' - R'$ we denote the grammar in which the productions in R' are removed from G'. For instance, for the augmented grammar

$$\{\oplus S \rightarrow a|b|c\}$$

a possible restriction is the set $\{S \rightarrow b, S \rightarrow c\}$, which corresponds to a selection and yields the grammar $\{\oplus S \rightarrow a\}$.

All restrictions are stored in an ordered list. The order $>$ between restrictions is as follows. This order is defined first by the *age* of the nonterminals in the restriction. Every copy rule introduces a new nonterminal in the grammar. The more recent the copy derivation was performed that introduced the nonterminal, the younger the nonterminal is considered to be. If the youngest nonterminal in R' is younger than the youngest variable in R, we define that $R' > R$. If two sets of productions contain nonterminals of the same age, the productions in both are sorted lexicographically. The ordered sets are then compared lexicographically.

Algorithm 1 GMiner (Grammar G, Restriction R, Database $\mathcal{D}$)

Apply all extensions of G to obtain G'
for each restriction $R' > R$ of G' **do**
  if $support(G' - R', \mathcal{D}) \geq minsup$ **then**
    GMiner ($G' - R'$, R', $\mathcal{D}$);

The reason for having this order is that we want to prevent that restrictions are performed in multiple orders. By only applying restrictions that are higher than the highest restriction that has already been applied, we prevent unnecessary duplicates in the search, even though the undecidability of grammar equivalence prevents us from avoiding equivalent grammars entirely.

Let us illustrate the consequences of this mechanism on two examples.

Supersets. Assuming given the grammar for supersets presented in the previous section, with $\mathcal{I} = \{a, b, c\}$, we obtain the following grammar after removing $S \rightarrow bS$:

$$\{\ominus \mathtt{S} \rightarrow \mathtt{a}\mathtt{S}|\mathtt{c}\mathtt{S}|\lambda\},$$

from this grammar we do no longer allow the removal of $S \rightarrow aS$, as the production that would be removed is lexicographically lower than $S \rightarrow bS$. If we would like to remove both $S \rightarrow aS$ and $S \rightarrow bS$, this is only possible by first removing $S \rightarrow aS$.

Ordered Induced Subtrees. Let us perform several refinements of the starting grammar

$$\{\oplus \mathtt{S} \rightarrow (i \bullet \mathtt{L}), \otimes \mathtt{L} \rightarrow \mathtt{L}(i \circ \mathtt{L}) \circ \mathtt{L}|\lambda \;:\; i \in \mathcal{I}\}$$

First, we perform a selection which removes all productions for S, except $\mathtt{S} \rightarrow (\mathtt{a}\mathtt{L})$:

$$\{\otimes \mathtt{S} \rightarrow (\mathtt{a} \bullet \mathtt{L}), \otimes \mathtt{L} \rightarrow \mathtt{L}(i \circ \mathtt{L}) \circ \mathtt{L}|\lambda \;:\; i \in \mathcal{I}\}$$

This grammar is extended into

$$\{\otimes \mathtt{S} \rightarrow (\mathtt{a}\mathtt{L}_1), \oplus \mathtt{L}_1 \rightarrow \mathtt{L}(i \bullet \mathtt{L}) \bullet \mathtt{L}|\lambda,$$
$$\otimes \mathtt{L} \rightarrow \mathtt{L}(i \circ \mathtt{L}) \circ \mathtt{L}|\lambda : i \in \mathcal{I}\}$$

In this grammar, we can remove productions for L_1, as L_1 is a younger variable than the variables L and S that occurred in the previously removed productions. For instance, we get

$$\{\otimes \mathtt{S} \rightarrow (\mathtt{a}\mathtt{L}_1), \otimes \mathtt{L}_1 \rightarrow \mathtt{L}(\mathtt{b} \bullet \mathtt{L}) \bullet \mathtt{L}, \\ \otimes \mathtt{L} \rightarrow \mathtt{L}(i \circ \mathtt{L}) \circ \mathtt{L} | \lambda : i \in \mathcal{I}\}$$

and extend this into

$$\{\otimes \mathtt{S} \rightarrow (\mathtt{a}\mathtt{L}_1), \otimes \mathtt{L}_1 \rightarrow \mathtt{L}(\mathtt{b}\mathtt{L}_2)\mathtt{L}_3, \\ \oplus \mathtt{L}_2 \rightarrow \mathtt{L}(i \bullet \mathtt{L}) \bullet \mathtt{L} | \lambda, \oplus \mathtt{L}_3 \rightarrow \mathtt{L}(i \bullet \mathtt{L}) \bullet \mathtt{L} | \lambda, \\ \otimes \mathtt{L} \rightarrow \mathtt{L}(i \circ \mathtt{L}) \circ \mathtt{L} | \lambda : i \in \mathcal{I}\}$$

In this grammar, if we perform a selection for L_3, and effectively create a sibling for b, we obtain

$$\{\otimes \mathtt{S} \rightarrow (\mathtt{a}\mathtt{L}_1), \otimes \mathtt{L}_1 \rightarrow \mathtt{L}(\mathtt{b}\mathtt{L}_2)\mathtt{L}_3, \\ \oplus \mathtt{L}_2 \rightarrow \mathtt{L}(i \bullet \mathtt{L}) \bullet \mathtt{L} | \lambda, \otimes \mathtt{L}_3 \rightarrow \mathtt{L}(\mathtt{c} \bullet \mathtt{L}) \bullet \mathtt{L}, \\ \otimes \mathtt{L} \rightarrow \mathtt{L}(i \circ \mathtt{L}) \circ \mathtt{L} | \lambda : i \in \mathcal{I}\}$$

In this grammar, our algorithm will not allow the removal of a production for L_2, as these productions are older than the productions that were removed last. If we want to create a tree in which b has a child, we have to first create that child before giving b a sibling. Effectively, it can be shown, we prevent that the same tree is created twice.

6 Parsing

An essential operation is the support evaluation of a set of grammars. There are multiple ways to evaluate the supports of a set of grammars. Most straightforward is to consider every grammar separately, and apply a standard chart parser, for instance, the Cocke-Younger-Kasami algorithm [21] or the Earley algorithm [8].

Simple optimizations of this approach are possible. In particular, in case the support is transaction-based, we can store for each grammar a *tid-set* of identifiers of transactions that it was able to parse. Restrictions of the grammar can only parse a subset of these transactions, so we can avoid that the algorithm tries to parse each string in the data repeatedly.

However, here we will show some more advanced possibilities to improve the efficiency of the search. There are two main ideas that we intend to exploit.

First, we can combine the parsing of multiple grammars. The grammars that need to be parsed in one call to the GMiner algorithm, are very similar. They only differ in their restrictions. While parsing the general grammar which includes all productions, we propose to keep track of which restrictions are used in all possible parses, and to use this information to determine the support counts of the restrictions. Similar ideas have been commonly used in other algorithms, for instance the pattern-growth approaches [3, 22, 23].

Second, we can combine the parsing of multiple strings. We will need to parse the same string many times, and therefore it can beneficial to store the data in such a way that multiple equal (sub)strings in the data are only parsed once.

We will illustrate these points for the traditional Cocke-Younger-Kasami algorithm for parsing context-free grammars. The CYK parser is one of the simplest chart parsers, which makes it more convenient to introduce the ideas behind our optimizations for this parser. However, the optimizations also apply in other chart parsers.

Parsing multiple grammars. The simplifying choice made in the CYK parser is that it only parses grammars in Chomsky normal form. We will first briefly repeat the essentials of the CYK algorithm. The CYK algorithm is based on the observation that for $|\alpha| \geq 2$:

$$A \stackrel{*}{\Rightarrow} \alpha \text{ iff } \exists \gamma, \beta : \alpha = \beta\gamma, \exists A \rightarrow BC \in P : B \stackrel{*}{\Rightarrow} \beta, C \stackrel{*}{\Rightarrow} \gamma,$$

which states that if we can divide a string into two parts β and γ, of which we know that they can be derived starting from nonterminals B and C, then if there is a production $A \rightarrow BC$, we know that the concatenation $\beta\gamma$ is derivable from A. This observation can be used in a dynamic programming algorithm by repeatedly considering substrings of increasing size until the entire string is covered: for each substring α, all possibilities to split it in two parts β and γ are scanned; if $B \stackrel{*}{\Rightarrow} \beta$ and $C \stackrel{*}{\Rightarrow} \gamma$, we check if there is a production $A \rightarrow BC$, and if so, the fact that $A \stackrel{*}{\Rightarrow} \alpha$ is stored. Typically, this information is maintained in a table, which is of size $O(n^2)$ for a string of length n.

Our extension of the CYK algorithm is based on the observation that for every substring and each nonterminal that can produce the substring, we can additionally determine the set of *forbidden* restrictions that, if applied, make it impossible to parse the string.

We can do this as follows. We start with the strings of length one. For each of these it is clear that a restriction is forbidden for nonterminal A and character σ iff the restriction contains the production $A \rightarrow \sigma$. Let us give every restriction R a unique identifier $id(R)$, and let $restr(P) = \{id(R) | P \in R\}$. Then we define that $forbid(A, \sigma) = restr(A \rightarrow \sigma)$.

Other substrings are processed in order of size using

the following relation:

$$forbid(A,\alpha) = \bigcap_{\substack{\beta,\gamma \text{ s.t.} \\ \alpha=\beta\gamma, \\ A\to BC\in P, \\ B\to\beta, C\to\gamma}} forbid(B,\beta)\cup forbid(C,\gamma)\cup restr(A\to BC)$$

This relation states that a restriction is only forbidden for $A \stackrel{*}{\Rightarrow} \alpha$ if it is forbidden in all possible derivations of $A \stackrel{*}{\Rightarrow} \alpha$ (therefore we have the outer intersection); we compute the forbidden groups for each derivation by determining all possible ways of splitting α in two parts, and considering the groups that are forbidden for at least one of the two parts (therefore we have the inner union); of course, the selection that is used to derive the nonterminal A from parts B and C is also forbidden, unless there are multiple restrictions in which the production $A \to BC$ is contained; in that case, however, the intersection removes both groups, as we intersect over all possible (A,α), (B,β) and G.

This relation is straightforwardly integrated in the CYK algorithm. If in the CYK algorithm we observe that $B \to \beta$, $C \to \gamma$ and $A \to BC \in P$, then we compute the union of the $forbid$ sets of (B,β), (C,γ) and $restr(A \to BC)$. The resulting set is intersected with the set that $(A,\beta\gamma)$ already had, if any.

Especially if the number of restrictions is small, the overhead of this computation is small. Depending on the number of elements in a set, we have the possibility to optimize this further by storing and maintaining the complement set of $forbid$ instead of $forbid$ itself.

We will illustrate this using the example of itemset mining. For reasons of simplicity, we assume that itemsets are encoded by separating the items with $*$ symbols. An example of an itemset is

$$*\mathtt{1}*\mathtt{10}*\mathtt{20}*$$

A starting grammar which searches for itemsets is then

$$\{\otimes\mathtt{S}\to\bullet\mathtt{R}, \otimes\mathtt{R}\to\mathtt{N}i\circ\mathtt{R}|*, \otimes\mathtt{N}\to *\,i\mathtt{N}|* : i\in\mathcal{I}\}$$

The initially locally copied grammar is

$$\{\otimes\mathtt{S}\to\mathtt{R}', \oplus\mathtt{R}'\to\mathtt{N}i\mathtt{R}|*, \otimes\mathtt{R}\to\mathtt{N}i\mathtt{R}|*, \otimes\mathtt{N}\to *i\mathtt{N}|* : i\in\mathcal{I}\},$$

which we can transform into CNF.

Assume that we are parsing the string $*\mathtt{1}*\mathtt{10}*\mathtt{20}*$. Then we will find out that every substring that starts and ends with a $*$ can be produced by N and R. For each such substring $forbid(N,\alpha) = forbid(R,\alpha) = \emptyset$. In the end, we have to compute the set $\mathcal{R}\backslash forbid(S, *\mathtt{1}*\mathtt{10}*\mathtt{20}*) = \{id(\oplus\mathtt{L}\to\mathtt{I_1F}), id(\oplus\mathtt{L}\to\mathtt{I_{10}F}), id(\oplus\mathtt{L}\to\mathtt{I_{20}F})\}$: only for these selections are we able to parse the string. The complexity of the parsing procedure is overall $O(n^3m)$, where n is the length of the string (7 in the example) and m is the number of items in the transaction (3 in the example). The naive solution, in which grammars are generated and parsed afterwards, would have complexity $O(n^3|\mathcal{I}|)$. For sparse itemset databases this is a significant improvement.

Similar situations can be expected in bottom-up chart parsers in other types of applications where the alphabet is large, but there is only a small number of sensible refinements in each example.

Parsing multiple strings. A problem when parsing general context free grammars is the potentially large size of the parse tables ($O(n^2)$ in the worst case). It is usually infeasible to parse long strings. A practical approach to deal with data consisting of long strings, is to move a window of a certain length over the data. We only parse the string within the window, and count either the number of parsable window positions, or the number of examples in which at least one parse happens.

In this approach, there is an obvious way to avoid redundant computations. If two windows overlap with eachother, they should also share the parts of their parse tables corresponding to common substrings. When the window is moved, we should therefore try to reuse relevant parts of the parse tables.

A more sophisticated solution is based on the observation that the same substring can also occur at several different positions in the data, and that we could also try to combine the parsing of these substrings. By storing all substrings of the database (possibly up to a certain length) in a *trie*, we can avoid that the same substring is parsed twice. A *trie* is a tree data structure in which every path from the root represents a string. Therefore, we can interchangeably refer to nodes and the strings represented by those nodes. The parse proceeds top-down through this tree, considering strings occurring in the tree in increasing size. Similar to the traditional CYK algorithm, nonterminals are computed for every substring by considering all possible ways of splitting the string into two parts; for each of the two parts it is extracted from the trie if they could be parsed. Usually, parsers do not construct such a trie, but as we have to parse the same string multiple times, the construction of such a trie may pay off.

After parsing all strings in the trie, we have found substrings that are derivable from S. From these substrings we need to compute the support of the grammar. There is an essential difference between the

position-based approach and the single string approach:

- for the transaction based approach, we need to associate to every string in the trie the identifiers of transactions in which this string occurs; the frequency is the size of the union of all identifier sets.

- for the position based approach, it suffices to store for every substring α in the trie the number of times ($count(\alpha)$) it occurs in the data. Let $\mathcal{S}$ be the set of all substrings in the trie that can be derived from starting symbol S, and let us remove from $\mathcal{S}$ all substrings that have a prefix that is also in $\mathcal{S}$, then one can show that $support(G, \mathcal{D}) = \sum_{\alpha \in \mathcal{S}'} count(\alpha)$, where $\mathcal{S}'$ is the subset of $\mathcal{S}$.

To speed-up the execution of the CYK algorithm on a trie, we can include suffix and prefix links in this data structure, similar to [6]. Sets of nonterminals, counts or transaction identifier sets, forbidden or allowed sets are associated to every node in the trie.

Other Optimizations. We developed several further optimizations in our method, of which we do not provide the full details here:

- dealing with *closed* grammars: in some grammars the removal of one production makes it impossible to 'reach' other productions. We could remove all such unreachable productions immediately, similar to what is common in *closed* pattern mining.

- reuse of parser information when refining: when we update a grammar by a copy or a deletion, we could reuse information of the previous parse of a string to only parse those parts again that are affected by the changed grammar.

A Final Note on Complexity. As pointed out, our algorithm lists grammars with polynomial delay: for each pattern we can determine in polynomial time how often it occurs in the data. Consequently, for any pattern domain that can be expressed as a grammar, our system provides a polynomial approach to find them, including tasks such as induced subtree mining. However, the $O(n^3)$ parsing procedure is still inferior to the $O(n)$ matching that is possible for specialized tasks such as itemset mining and induced subtree mining.

7 Experiments

In this section we investigate the practical use of our system in two types of experiments.

Performance Evaluation. In our first experiments we evaluate the runtime of our algorithms on UCI

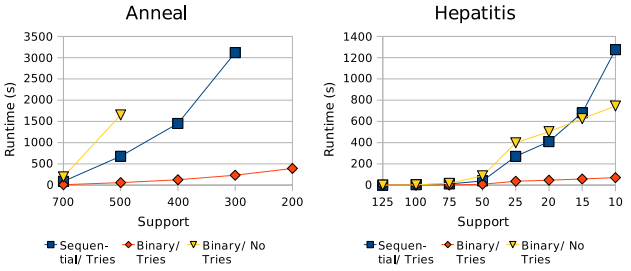

Figure 2: A comparison of grammar mining algorithms on two UCI datasets; Hepatitis contains 137 transactions over 24 items; Anneal contains 812 transactions over 42 items.

data, as shown in Figure 2[1]. Experiments were performed on a machine with an Intel Pentium D 2.80GHz CPU and 2GB of memory. We use two algorithms that we implemented in C++, both of which share as much source code as possible. One algorithm implements a parsing procedure on a trie (thus allowing re-use of parses between examples); the other algorithm does not build a trie, but parses each string in the input data individually. Both implementations extend a standard CYK parser, and thus transform grammars into CNF. They both include the bottom-up computations of forbidden restrictions. As test-case we used the problem of frequent itemset mining, for which we tried the two different formulations of this problem (see page 6). We used entire transaction-based support. We also ran the LCM itemset miner on this data [18]; LCM finished within < 2s on each of the datasets for all support thresholds.

Comparing the runtime of our algorithm with LCM, it is not surprising that our system is much slower. If we compare the two representations of the itemset mining task, we see that the representation for binary data is faster. This reflects our intuition: the grammar for the binary representation involves a smaller alphabet, a smaller number of productions, and is not ambiguous. Comparing the binary representation with and without the use of a trie-structure, we see that parsing on the trie-structure is beneficial on all datasets that we tried.

We performed a similar experiment for the problem of mining ordered induced subrees. In this experiment we compared our approach with the FREQT algorithm [2]. Again, we found that our algorithm found the same subtrees as this specialized algorithm, but at a higher computational cost: for instance, on the artificially generated D2 dataset (for details, see [4]),

[1]These datasets are available from: `http://www.cs.kuleuven.be/~dtai/CP4IM/`

which contains 10000 trees of on average 18.4 nodes, we require 196s to find all trees for a support threshold of 2000; FREQT requires 3s.

Discovering Patterns in Reality Mining Data. Our second type of experiment was performed on a log file of telephone calls for one person in the Reality Mining data set (`http://reality.media.mit.edu/`). Our aim is to demonstrate the use of our approach for an untraditional pattern mining problem. The dataset contains one long string in $(TD\mathcal{N})^*$, where $T = \{\texttt{ShortMessage}, \texttt{VoiceCall}\}$, $D = \{\texttt{Incoming}, \texttt{Outgoing}, \texttt{MissedCall}\}$; $\mathcal{N}$ is the set of contact identifiers. We analyzed data for person 1566 using this grammar with $P = P_\otimes \cup P_\ominus \cup P_\oplus$:

$$\begin{aligned} P_\otimes &= \cup_{\sigma\in\mathcal{N}}\{\otimes\texttt{S} \to \sigma\texttt{YG}\sigma\texttt{Z}, \otimes\texttt{A} \to \sigma\} \\ &\quad \cup_{\sigma_2\in T}\{\otimes\texttt{B} \to \sigma_2\} \cup_{\sigma_3\in D}\{\otimes\texttt{C} \to \sigma_3\} \\ P_\ominus &= \{\ominus\texttt{G} \to \lambda, \ominus\texttt{G} \to \texttt{ABC}, \ominus\texttt{G} \to \texttt{ABCABC}\} \\ P_\oplus &= \cup_{\sigma_1\in\texttt{T},\sigma_2\in\texttt{D}}\{\oplus\texttt{Y} \to \sigma_1\sigma_2, \oplus\texttt{Z} \to \sigma_1\sigma_2\} \end{aligned}$$

This grammar finds repeated interactions with the same contact person. An example of a grammar we find is:

$$\begin{aligned} &P_\otimes \cup P_\ominus \cup \{\texttt{Y} \to \texttt{ShortMessage Incoming}\} \\ &\cup\{Z \to \texttt{VoiceCall Outgoing}\}. \end{aligned}$$

8 Related Work

As pointed out, the task that we studied is related substring mining [19], subsequence mining [17, 24, 14] and subtree mining [25, 2], in which one aims at finding patterns that are frequent within a fixed domain of patterns. Our algorithm is more general than these special-purpose algorithms — for instance, by allowing the user to express both embedded and induced subtree mining. In some cases this comes at the price of a provably worse performance.

The idea of providing a user the ability to specify pattern domains is related to the idea of constraint-based mining, for instance, substring mining under constraints [1, 9]. Existing algorithms mainly focus on constraining substrings or subtrees under consideration. By using grammars as pattern domain, we can effectively constrain how patterns match with data; this is not possible in existing systems.

In our approach we use some ideas from for multi-relational pattern mining [7, 11]. However, by focusing on string domains, our approach is theoretically more efficient than these approaches.

Grammar *learning* is a popular topic in the machine learning and natural language processing communities, for instance [5, 15, 13]. These related approaches aim at finding one grammar that parses a set of examples as accurately as possible; the grammar should be a *model* for the data. In our work we did not aim at finding a *global* model; we took a pattern mining point of view, aiming at finding sets of *local* patterns, which only describe subsets of examples.

9 Conclusions and Future Work

The main contribution of this paper is that we proposed the problem of grammar mining as a generalization of many pattern mining tasks. Furthermore we proposed the development of bias specification languages which allow users to specify wide ranges of pattern domains. We provided one example of a specification language and an algorithm for finding grammars within the specified domain.

We verified experimentally that the approach works as expected. Indeed, we were able to use our algorithm to find all patterns discovered by other systems, and succesfully achieved our goal of developing a usable general pattern mining system for string data. We verified the computational performance of our system, although the main aim of this study was not to improve specialized systems in terms of performance. As expected, we found that its performance was not competitive with specialized systems; however, the proposed optimizations improved scalability as desired. In a case study we illustrated the use of our system.

There are several possibilities for further work in this direction.

First, there are many possibilities to improve performance. As we choose general CFG grammars as pattern domain in this work, we were forced to use relatively inefficient chart parsers. For many types of grammars more efficient parsers are known. How to exploit such parsers, possibly in a semi-automatic way, is an important question to improve the scalability of our method.

A second question is how to extend the approach to structured domains such as graphs, where the order between edges in the pattern may be different from the order in the data. Our current approach is limited to ordered domains.

A third question is how to visualize patterns. In some cases it may be desirable to print a grammar in a simplified form; for instance, a grammar for the itemset $\{1, 2, 3\}$ could be printed more 'prettily' as `123`. Such pretty printing may be achieved by attaching printing instructions to derivation rules, similar to instructions that are executed by compilers when parsing a program.

A fourth problem is the development of good bias specification languages. In this paper we proposed one possible approach. Other specification languages may be desirable for other tasks. In the area of Inductive Logic Programming (ILP), for instance, several mechanisms have been studied for specifying language bias on conjunctive formulas. While these mechanisms are not

directly applicable in the grammar mining setting that we studied here, the analogy of our approach to ILP suggests that other languages deserve study. In particular, with certain application domains in mind, such as bioinformatics, intuitive approaches may be obtained by studying special types of grammars, such String Variable Grammars [16] or Basic Gene Grammars [20].

Finally, we wish to integrate our approach in a scripting language such as Python. Using such a system, we hope to enable a wider range of case studies to show the applicability of our approach.

References

[1] H. Albert-Lorincz and J.-F. Boulicaut. Mining frequent sequential patterns under regular expressions: a highly adaptive strategy for pushing constraints. In *Proceedings of the Third SIAM International Conference on Data Mining (SDM)*, pages 316–320. SIAM, 2003.

[2] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In *SDM*, 2002.

[3] H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Improving the efficiency of inductive logic programming through the use of query packs. In *Journal of Artificial Intelligence Research*, volume 16, pages 135–166. AAAI Press, 2002.

[4] Y. Chi, R. R. Muntz, S. Nijssen, and J. N. Kok. Frequent subtree mining - an overview. *Fundam. Inform.*, 66(1-2):161–198, 2005.

[5] C. De la Higuera, P. Adriaans, M. Van Zaanen, and J. Oncina, editors. *Proceedings of the ECML/PKDD Workshop and Tutorial on Learning Context-Free Grammars*, 2003.

[6] L. De Raedt, M. Jaeger, S. D. Lee, and H. Mannila. A theory of inductive query answering (extended abstract). In *Proceedings of the Second IEEE International Conference on Data Mining (ICDM)*, pages 123–130. IEEE Press, 2002.

[7] L. Dehaspe and H. Toivonen. Discovery of frequent Datalog patterns. In *Data Mining and Knowledge Discovery*, volume 3, pages 7–36. Kluwer Acadamic Publishers, 1999.

[8] J. Earley. An efficient context-free parsing algorithm. 13:94–102, 1970.

[9] M. N. Garofalakis, R. Rastogi, and K. Shim. SPIRIT: sequential pattern mining with regular expression constraints. In *Proceedings of the 25th International Conference on Very Large Databases (VLDB)*, pages 223–234. Morgan Kaufmann Publishers, 1998.

[10] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM Press, 2000.

[11] M. Hasan, V. Chaoji, S. Salem, N. Parimi, and M. Zaki. DMTL: A generic data mining template library. In *Proceedings of the Workshop on Library-Centric Software Design*, 2005.

[12] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. In *J. Log. Program.*, volume 19/20, pages 629–679, 1994.

[13] C. G. Neville-Manning and I. H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[14] J. Pei, J. Han, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 215–224. IEEE Press, 2001.

[15] Y. Sakakibara. Efficient learning of context-free grammars from positive structural examples. In *Information and Computation*, volume 97, pages 23–60, 1992.

[16] D. B. Searls. String variable grammar: A logic grammar formalism for the biological language of DNA. *J. Log. Programming*, 24:73–102, 1995.

[17] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proceedings of the 5th International Conference on Extending Database Technology (EDBT)*, volume 1057 of *Lecture Notes in Computer Science*, pages 3–17. Springer-Verlag, 1996.

[18] T. Uno, M. Kiyomi, and H. Arimura. Lcm ver.3: collaboration of array, bitmap and prefix tree for frequent itemset mining. In *OSDM '05: Proceedings of the 1st international workshop on open source data mining*, pages 77–86, 2005.

[19] J. Vilo. Discovering frequent patterns from strings. Technical Report C-1998-9, Department of Computer Science, University of Helsinki, 1998.

[20] S. wai Leung, C. Mellish, and D. Robertson. Basic gene grammars and DNA-chartparser for language processing of escherichia coli promoter DNA sequences. *Bioinformatics*, 17:226–236, 2001.

[21] D. Wood. *Theory of Computation*. John Wiley and Sons, 1987.

[22] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *Proceedings of the Second IEEE International Conference on Data Mining (ICDM)*, pages 721–724. IEEE Press, 2002.

[23] X. Yan and J. Han. CloseGraph: Mining closed frequent graph patterns. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 286–295. ACM Press, 2003.

[24] M. J. Zaki. Efficient enumeration of frequent sequences. In *Proceedings of the Seventh International Conference on Information and Knowledge Management*, pages 68–75. ACM Press, 1998.

[25] M. J. Zaki. Efficiently mining frequent trees in a forest: Algorithms and applications. *IEEE Trans. Knowl. Data Eng.*, 17(8):1021–1035, 2005.