Mining Sequential Patterns with Constraints in Large Databases

Jian Pei
State University of New York at Buffalo
jianpei@cse.buffalo.edu

Jiawei Han
Univ. of Illinois at Urbana-Champaign
hanj@cs.uiuc.edu

Wei Wang
Fudan University
weiwang1@fudan.edu.cn

ABSTRACT

Constraints are essential for many sequential pattern mining applications. However, there is no systematic study on *constraint-based sequential pattern mining*. In this paper, we investigate this issue and point out that the framework developed for constrained frequent-pattern mining does not fit our missions well. An extended framework is developed based on a *sequential pattern growth* methodology. Our study shows that constraints can be effectively and efficiently pushed deep into sequential pattern mining under this new framework. Moreover, this framework can be extended to constraint-based structured pattern mining as well.

1. INTRODUCTION

There have been many studies on efficient sequential pattern mining and its applications (e.g. [2, 10, 5, 9, 11]). Sequential pattern mining algorithms, in general, can be categorized into three classes: (1) Apriori-based, horizontal formatting method, with GSP [10] as its representative; (2) Apriori-based, vertical formatting method, such as SPADE [11]; and (3) projection-based pattern growth method, such as PrefixSpan [9].

For effectiveness and efficiency considerations, constraints are essential in many data mining applications. In the context of constraint-based sequential pattern mining, Srikant and Agrawal [10] generalize the scope of sequential pattern mining [2] to include time constraints, sliding time windows, and user-defined taxonomy. Mining frequent episodes in a sequence of events studied by Mannila, et al. [5] can also be viewed as a constrained mining problem, since episodes are essentially constraints on events in the form of acyclic graphs. Garofalakis et al. [3] propose regular expressions as constraints for sequential pattern mining and develop a family of SPIRIT algorithms, while members in the family achieve various degrees of constraint enforcement. They use relaxed constraints having nice properties (e.g. anti-monotonicity) to filter out some unpromising patterns/candidates in their early stage.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CIKM'02, November 4–9, 2002, McLean, Virginia, USA.
Copyright 2002 ACM 1-58113-492-4/02/0011 ...$5.00.

The above interesting studies handle a few scattered classes of constraints. However, two problems remain: (1) many practical constraints are not covered, and (2) there lacks a systematic method to push various constraints into the mining process.

To motivate this study, let us consider the following example. To characterize a new disease, researchers may want to find sequential patterns about symptoms, such as "*finding patterns with constraint of* $2-7$ *days of cough followed by fever in range of* $37.5-39C$ *for* $2-5$ *days with average temperature of* $38\pm0.2C$*, and all these symptoms appear within 2 weeks.*". A pattern found could be "*cough 5 days and fever 4 days with strong headache.*" This mining query contains a few constraints, involving sequences containing certain constants, and with average functions, etc. None of the previously developed constraint-based sequential pattern mining methods can handle all these constraints. Moreover, it is unclear how to incorporate all constraints in the mining process.

In this paper, we conduct a systematic study on constraint-based sequential pattern mining. First, various kinds of constraints are classified in two orthogonal ways, based on their application semantics (Section 2) and their roles in sequential pattern mining (Section 3), respectively. The later scheme largely follows the conventional constraint-based frequent pattern mining framework. Unfortunately, some commonly encountered sequence-based constraints, such as regular expression constraints, are neither monotonic, nor anti-monotonic, nor succinct. Instead of patching the classical framework, a new framework, called Prefix-growth, is built based on a *prefix-monotone* property (Section 4). Interestingly, all the monotonic and anti-monotonic constraints, as well as regular expression constraints, are *prefix-monotone*, and can be pushed deep into a prefix-growth-based mining algorithm. Moreover, some tough aggregate constraints, such as those involving average or general sum, can also be pushed deep into a slightly revised prefix-growth mining process (Section 5). A performance study is conducted which demonstrates that constraint-based mining prunes a large search space effectively in sequential pattern mining, and prefix-growth is more efficient than other constraint-based sequential pattern mining algorithms studied before (Section 6).

2. SEQUENTIAL PATTERN MINING AND CATEGORIES OF CONSTRAINTS

Let $I = \{x_1, \ldots, x_n\}$ be a set of **items**, each possibly being associated with a set of **attributes**, such as value,

price, profit, calling distance, period, etc. The value on attribute A of item x is denoted as $x.A$. An **itemset** is a non-empty subset of items, and an itemset with k items is called a **k-itemset**.

A **sequence** $\alpha = \langle X_1 \cdots X_l \rangle$ is an ordered list of itemsets. An itemset X_i ($1 \leq i \leq l$) in a sequence is called a **transaction**, a term originated from analyzing customers' shopping sequences in a transaction database, such as in [10]. A transaction X_i may have a special attribute, *times-tamp*, denoted as $X_i.time$, which registers the time when the transaction was executed. As a notational convention, for a sequence $\alpha = \langle X_1 \cdots X_l \rangle$, $X_i.time < X_j.time$ for $1 \leq i < j \leq l$.

The number of transactions in a sequence is called the **length** of the sequence. A sequence α with length l is called an **l-sequence**, denoted as $len(\alpha) = l$. The i-th itemset is denoted as $\alpha[i]$. Following the convention in [10], an item can occur at most once in an itemset, but can occur multiple times in various itemsets in a sequence.

A sequence $\alpha = \langle X_1 \ldots X_n \rangle$ is called a **subsequence** of another sequence $\beta = \langle Y_1 \ldots Y_m \rangle$ ($n \leq m$), and β a **super-sequence** of α, denoted as $\alpha \sqsubseteq \beta$, if there exist integers $1 \leq i_1 < \ldots < i_n \leq m$ such that $X_1 \subseteq Y_{i_1}, \ldots, X_n \subseteq Y_{i_n}$.

A **sequence database** SDB is a set of 2-tuples (sid, α), where sid is a **sequence-id** and α a sequence. A tuple (sid, α) in a sequence database SDB is said to **contain** a sequence γ if γ is a subsequence of α. The number of tuples in a sequence database SDB containing sequence γ is called the **support** of γ, denoted as $sup(\gamma)$.

Given a positive integer min_sup as the *support threshold*, a sequence γ is a **sequential pattern** in sequence database SDB if $sup(\gamma) \geq min_sup$. The **sequential pattern mining** problem is to find the *complete* set of sequential patterns with respect to a given sequence database SDB and a support threshold min_sup.

For example, Table 1 shows a sequence database SDB with four sequences. The first contains three transactions (itemsets) (i.e., length = 3): $\{a\}$, $\{b, c\}$ and $\{e\}$. For brevity, the brackets are omitted if a transaction has only one item.

Sequence_id	Sequence
10	$\langle a(bc)e \rangle$
20	$\langle e(ab)(bc)dd \rangle$
30	$\langle c(aef)(abc)dd \rangle$
40	$\langle addcb \rangle$

Table 1: Sequence database SDB.

Sequence $\langle (ab)d \rangle$ is a subsequence of both the second sequence, $\langle e(ab)(bc)dd \rangle$, and the third one, $\langle c(aef)(abc)dd \rangle$. So, if the support threshold $min_sup = 2$, $\langle (ab)d \rangle$ is a sequential pattern.

Like many frequent pattern mining problems [1, 4], there are two major difficulties in sequential pattern mining: (1) *effectiveness*: mining may return a huge number of patterns, many of which could be uninteresting to users, and (2) *efficiency*: it often takes substantial processing power for mining the complete set of sequential patterns in a large sequence database. Constraint-based mining may overcome both difficulties since constraints usually represent user's interest and focus, which confines the patterns to be found to a particular set of conditions. Moreover, if constraints can be pushed deep into the mining process, it is likely to achieve efficiency since the search can be more focused. This motivates the study of constraint-based mining of sequential patterns.

Let **constraint** C for a sequential pattern α be a boolean function $C(\alpha)$. The **problem of constraint-based sequential pattern mining** is to find the complete set of sequential patterns, denoted as $SAT(C)$, satisfying a given constraint C.

Constraints can be examined and characterized from different points of views. We examine them first from the application point of view in this section and then from the constraint-pushing point of view in the next section, and build up linkages between the two by a thorough study of constraint-based sequence mining.

From the application point of view, we present the following seven categories of constraints based on the semantics and the forms of the constraints. Although this is by no means complete, it covers most of the interesting constraints in applications.

Constraint 1. An *item constraint* specifies what are the particular individual or groups of items that should or should not be present in the patterns. It is in the form of $C_{item}(\alpha) \equiv (\varphi i : 1 \leq i \leq len(\alpha),\ \alpha[i]\ \theta\ V)$, or $C_{item}(\alpha) \equiv (\varphi i : 1 \leq i \leq len(\alpha),\ \alpha[i] \cap V \neq \emptyset)$, where V is a subset of items, $\varphi \in \{\forall, \exists\}$ and $\theta \in \{\subseteq, \not\subseteq, \supseteq, \not\supseteq, \in, \notin\}$.[1]

For example, when mining sequential patterns over a web log, a user may be interested in only patterns about visits to online bookstores. Let B be the set of online bookstores. The corresponding item constraint is $C_{bookstore}(\alpha) \equiv (\forall i : 1 \leq i \leq len(\alpha), \alpha[i] \subseteq B)$. □

Constraint 2. A *length constraint* specifies the requirement on the length of the patterns, where the length can be either the number of occurrences of items or the number of transactions. Length constraints can also be specified as the number of distinct items, or even the maximal number of items per transactions.

For example, a user may want to find only long patterns (e.g., at least 50 transactions) in bio-sequence analysis. Such a requirement can be expressed by a length constraint $C_{len}(\alpha) \equiv (len(\alpha) \geq 50)$. □

Constraint 3. A *super-pattern constraint* is in the form of $C_{pat}(\alpha) \equiv (\exists \gamma \in P \text{ s.t. } \gamma \sqsubseteq \alpha)$, where P is a given set of patterns, i.e., to find patterns that contain a particular set of patterns as sub-patterns.

For example, an analyst may like to find the sequential patterns that contain first buying a PC and then a digital camera, the constraint can be expressed as $C_{pat}(\alpha) \equiv \langle (PC)(digital_camera) \rangle \sqsubseteq \alpha$. □

Constraint 4. An *aggregate constraint* is the constraint on an aggregate of items in a pattern, where the aggregate function can be sum, avg, max, min, standard deviation, etc.

For example, a marketing analyst may like sequential patterns where the average price of all the items in each pattern is over \$100. □

Constraint 5. A *regular expression constraint* C_{RE} is a constraint specified as a regular expression over the set of items using the established set of regular expression operators, such as disjunction and Kleene closure. A sequential pattern satisfies C_{RE} if and only if the pattern is accepted by its equivalent deterministic finite automata.

[1] For brevity, we omit the strict operators (e.g., $\subset, \supset$) in our discussion. However, the same principles can be applied to them.

For example, to find sequential patterns about a Web click stream starting from Yahoo's home page and reaching hotels in New York city, one may use regular expression constraint *Travel* (*New York* | *New York City*)(*Hotels* | *Hotels and Motels* | *Lodging*), where "|" stands for disjunction. The concept of regular expression constraint was first proposed in [3]. □

In some applications, one may want to have constraints on the duration of the patterns, i.e., events happening within a certain duration.

Constraint 6. A *duration constraint* is defined only in sequence databases where each transaction in every sequence has a time-stamp. It requires that the pattern appears frequently in the sequence database such that the time-stamp difference between the first and last transactions in the pattern must be longer or shorter than a given period. Formally, a duration constraint is in the form of $Dur(\alpha)\ \theta\ \Delta t$, where $\theta \in \{\leq, \geq\}$ and Δt is a given integer. A sequence α satisfies the constraint if and only if $|\{\beta \in SDB | \exists 1 \leq i_1 < \cdots < i_{len(\alpha)} \leq len(\beta)$ s.t. $\alpha[1] \sqsubseteq \beta[i_1], \ldots, \alpha[len(\alpha)] \sqsubseteq \beta[i_{len(\alpha)}]$, and $(\beta[i_{len(\alpha)}].time - \beta[i_1].time)\ \theta\ \Delta t\}| \geq min_sup$. □

In some other applications, the gap between adjacent transactions in a pattern may be important.

Constraint 7. A *gap constraint* set is defined only in sequence databases where each transaction in every sequence has a timestamp. It requires that the pattern appears frequently in the sequence database such that the timestamp difference between every two adjacent transactions must be longer or shorter than a given gap. Formally, a gap constraint is in the form of $Gap(\alpha)\ \theta\ \Delta t$, where $\theta \in \{\leq, \geq\}$ and Δt is a given integer. A sequence α satisfies the constraint if and only if $|\{\beta \in SDB | \exists 1 \leq i_1 < \cdots < i_{len(\alpha)} \leq len(\beta)$ s.t. $\alpha[1] \sqsubseteq \beta[i_1], \ldots, \alpha[len(\alpha)] \sqsubseteq \beta[i_{len(\alpha)}]$, and for all $1 < j \leq len(\alpha)$, $(\beta[i_j].time - \beta[i_{j-1}].time)\ \theta\ \Delta t\}| \geq min_sup$. □

Among the constraints listed above, duration constraints and gap constraints are *support-related*, i.e. they are applied to confine how a sequence matches a pattern. To find whether a sequential pattern satisfies these constraints, one needs to examine the sequence databases. For other constraints, whether the constraint is satisfied can be determined by the frequent patterns themselves, without referring to the support counting process.

3. A CLASSICAL FRAMEWORK OF CHARACTERIZATION OF CONSTRAINTS

In recent studies of constrained frequent pattern mining [6, 7, 8], constraints are characterized based on the notion of monotonicity, anti-monotonicity, succinctness, and whether they can be transformed into these categories if they do not belong to them. This has become a classical framework for constraint-based frequent pattern mining. "*Can we extend this framework and solve the constraint-based sequential pattern mining problem?*"

A constraint C_A is **anti-monotonic** if a sequence α satisfying C_A implies that every non-empty subsequence of α also satisfies C_A. A constraint C_M is **monotonic** if a sequence α satisfies C_M implies that every super-sequence of α also satisfies C_M. The basic idea behind **succinct constraint** is that, with such a constraint, one can explicitly and precisely generate all the sets of items satisfying the constraint without recourse to a generate-everything-and-test approach. A succinct constraints is specified using a precise "formula". According to the "formula", one can generate all the patterns satisfying a succinct constraint. There is no need to iteratively check the constraint in the mining process. Limited by space, we omit the formal definitions here.

For example, length constraint $C_{len}(\alpha) \equiv len(\alpha) \leq 10$ and duration constraint $Dur(\alpha) \leq 30$ are anti-monotonic, while super-pattern constraint and the duration constraint $Dur(\alpha) \geq 30$ are monotonic. It is easy to show that item constraints, length constraints and super-pattern constraints are all succinct.

Based on the above definition, the anti-monotonic, monotonic and succinct characteristics of some commonly used constraints for sequential pattern mining are shown in Table 2.

From Table 2, one can see that the classical constraint-pushing framework [6] based on anti-monotonicity, monotonicity, and succinctness can be applied to a large class of constraints. Thus the corresponding constraint-pushing strategy can be integrated easily into any one of sequential pattern mining algorithms, such as GSP, SPADE, and PrefixSpan. However, some important classes of constraints, such as RE (regular expressions), average(i.e., $avg(\alpha)\ \theta\ v$, where $\theta \in \{\leq, \geq\}$), and g_sum (i,e., *sum* of positive and negative values), do not fit into this framework.

This problem, w.r.t. commonly used regular expression constraints, has been pointed out by Garofalakis, et al. [3]. They provide a solution of a set of four SPIRIT algorithms, each pushing a stronger relaxation of regular expression constraint $\mathcal{R}$ than its predecessor in the pattern mining loop. The above method, though interesting, does not elegant and systematic for all kinds of constraints. Can we handle those *ugly* constraints in a nice and elegant way? This is the theme of the next section.

4. A NEW FRAMEWORK: MINING SEQUENTIAL PATTERNS WITH PREFIX-MONOTONE CONSTRAINTS

The classical frequent pattern mining framework is based on the anti-monotonic Apriori property [1]. To handle *regular expression*-like *"ugly"* but popularly encountered constraints effectively, one needs to jump out of this framework to re-examine the basic properties of constraints and their evaluation framework.

4.1 Prefix-Monotone Property

Let R be an order of items in a sequence database. Since the item ordering in the same transaction is unimportant to sequential patterns, it is convenient to assume that all items in a transaction are written with respect to the order R. For example, let R be the alphabetical order. A sequence should be written in the form of $\langle(ade)(bc)\rangle$ instead of $\langle(dae)(cb)\rangle$. The fact that item x is before item y according to order R is denoted as $x \prec y$.

Given a sequence $\alpha = \langle X_1 \cdots X_n\rangle$, sequence $\beta = \langle X_1 \cdots X_k Y\rangle$ is called a **prefix** of α if (1) $k < n$, (2) $Y \subseteq X_{k+1}$, and (3) $\forall y \in Y, \forall z \in (X_{k+1} - Y)$, $y \prec z$. For example, sequence $\beta = \langle(abc)(ac)\rangle$ is a prefix of sequence $\alpha = \langle(abc)(acd)(bef)\rangle$ but sequence $\gamma = \langle(abc)(ad)\rangle$ is not a prefix of α. Here, the alphabetical order is used.

A constraint C_{pa} is called **prefix anti-monotonic** if for each sequence α satisfying the constraint, so does every prefix of α. A constraint C_{pm} is called **prefix monotonic** if for each sequence α satisfying the constraint, so does every sequence having α as a prefix. A constraint is called

Constraint		Anti-mono	Mono	Succ
Item	$C_{item}(\alpha) \equiv (\forall i : 1 \leq i \leq len(\alpha),\ \alpha[i]\theta V)\ (\theta \in \{\subseteq, \supseteq\})$	Yes	No	Yes
	$C_{item}(\alpha) \equiv (\forall i : 1 \leq i \leq len(\alpha),\ \alpha[i] \cap V \neq \emptyset)$	Yes	No	Yes
	$C_{item}(\alpha) \equiv (\exists i : 1 \leq i \leq len(\alpha),\ \alpha[i]\theta V)\ (\theta \in \{\subseteq, \supseteq\})$	No	Yes	Yes
	$C_{item}(\alpha) \equiv (\exists i : 1 \leq i \leq len(\alpha),\ \alpha[i] \cap V \neq \emptyset)$	No	Yes	Yes
Length	$len(\alpha) \leq l$	Yes	No	Yes
	$len(\alpha) \geq l$	No	Yes	Yes
Super-pattern	$C_{pat}(\alpha) \equiv (\exists \gamma \in P \text{ s.t. } \gamma \sqsubseteq \alpha)$	No	Yes	Yes
Simple aggregates	$max(\alpha) \leq v),\ min(\alpha) \geq v$	Yes	No	Yes
	$max(\alpha) \geq v),\ min(\alpha) \leq v$	No	Yes	Yes
	$sum(\alpha) \leq v$ (with non-negative values)	Yes	No	No
	$sum(\alpha) \geq v$ (with non-negative values)	No	Yes	No
Tough aggregates	g_sum: $sum(\alpha)\ \theta\ v,\ \theta \in \{\leq, \geq\}$ (with positive and negative values)	No	No	No
	average: $avg(\alpha)\ \theta\ v$	No	No	No
RE	(Regular Expression) ¶	No	No	No
Duration	$Dur(\alpha) \leq \Delta t$	Yes	No	No
	$Dur(\alpha) \geq \Delta t$	No	Yes	No
Gap	$Gap(\alpha)\ \theta\ \Delta t\ (\theta \in \{\leq, \geq\})$	Yes	No	No

Table 2: Characterization of commonly used constraints. (¶ In general, a regular expression (RE) constraint is not necessarily anti-monotonic, monotonic, or succinct, though there are cases that some of them are.)

prefix-monotone if it is prefix anti-monotonic or prefix monotonic.

Clearly, if β is a prefix of α, β is also a subsequence of α. Intuitively, an anti-monotonic constraint is (trivially) prefix anti-monotonic. A monotonic constraint is (trivially) prefix monotonic. For example, the length constraint $len(\alpha) \leq 10$ is anti-monotonic. It must also be prefix anti-monotonic. This is because if the length of a sequence α is no more than 10, the length of every prefix of α must be no more than 10 as well. Similarly, $len(\alpha) \geq 10$ is prefix monotonic since if the length of any prefix of α is no less than 10, α must be no less than 10 as well.

A succinct constraint is not necessarily prefix anti-monotonic or prefix monotonic. However, since succinct constraints can be pushed deep directly into the mining process (no matter which sequential pattern mining method is applied), the pushing of such constraints will not be analyzed further in our discussion.

Now, let's examine regular expression constraints. A well-known result from the formal language theory is that for every regular expression E, there exists a deterministic finite automata M_E such that M_E accepts exactly the language generated by E.

Given a regular expression E, let M_E be the corresponding (deterministic finite) automata. Let α be a sequence. α is called **legal** w.r.t. E if a state in M_E can be reached following α. From a regular expression constraint E, we can derive a constraint L_E such that a sequence α satisfies L_E if and only if α is legal w.r.t. E. Clearly, for each sequence α satisfying the regular expression constraint E, every prefix of α must be legal w.r.t. E. Furthermore, for each sequence β legal w.r.t. E, every prefix of β must also be legal w.r.t. E. That is, *the constraint L_E on legal prefix w.r.t. E is prefix anti-monotonic.* Based on the above discussion, we have the following statement.

THEOREM 4.1. *All the commonly used constraints discussed in Section 2, except for* g_sum *and* average*, have prefix-monotone property.* □

Theorem 4.1 indicates that prefix-monotone property covers more constraints commonly used than traditional anti-monotonic and monotonic properties, since prefix-monotone property is weaker than anti-monotone and monotone properties. All anti-monotonic or monotonic constraints are prefix-monotonic. One may wonder whether that means mining with prefix-monotone property is less efficient than the classical anti-monotonicity-based Apriori methods? Fortunately and surprisingly, the answer is no.

4.2 Pushing Prefix-Monotone Constraints into Sequential Pattern Mining

First, we introduce the concept of *projected database*. For sequence $\alpha \sqsubseteq \beta$, sequence γ is said the **projection** of β w.r.t. α if (1) $\gamma \sqsubseteq \beta$, (2) α is a prefix of γ, and (3) there exists no proper super-sequence γ' of γ such that $\gamma' \sqsubseteq \beta$ and γ' also has α as a prefix. Projection is also denoted as $\gamma = \beta/\alpha$. For example, if $\alpha = \langle bc \rangle$, $\beta = \langle (abc)d(ace)f \rangle$, then $\gamma = \beta/\alpha = \langle b(ce)f \rangle$.

Give a sequence database SDB and a sequence α. The **α-projected database**, denoted as $SDB|_\alpha$, is the set of projections of sequences in SDB having α as a subsequence, i.e., $SDB|_\alpha = \{\gamma | \gamma = \beta/\alpha, \beta \in SDB \wedge \alpha \sqsubseteq \beta\}$.[2]

Now, let us examine an example of constraint pushing.

Let the sequence database SDB be Table 1, and the task be mining sequential patterns with a regular expression constraint $C = \langle a{*}\{bb|(bc)d|dd\} \rangle$ and support threshold $min_sup = 2$. The mining can be performed in the following steps.

Step 1: find length-1 patterns and remove irrelevant sequences. Scanning SDB once, patterns $\langle a \rangle$, $\langle b \rangle$, $\langle c \rangle$, $\langle d \rangle$, and $\langle e \rangle$ are identified as length-1 patterns. Infrequent items, such as f, is removed. Also, in the same scan, the sequences that contain no subsequence satisfying the constraint should be removed, such as the first sequence, $\langle a(bc)e \rangle$, which fails the regular expression constraint.

Step 2: divide the set of sequential patterns into subsets without overlap. The complete set of sequential patterns can be divided into five subsets without overlap according

[2]Since α appears as prefix in every sequence in $SDB|_\alpha$, for brevity, we can omit the occurrences of α as prefixes in $SDB|_\alpha$. For a sequence $\beta \in SDB|_\alpha$, we only record the **suffix** β'. If the last transaction of α and the first transaction of β' are in the same transaction of β, then a symbol "_" is put in the first transaction of β'. Therefore, we have $\beta = \alpha \cdot \beta'$.

to the set of length-1 sequential patterns: (1) those with prefix $\langle a \rangle$; (2) those with prefix $\langle b \rangle$; ...; and (5) those with prefix $\langle e \rangle$. Since only patterns with prefix $\langle a \rangle$ may satisfy the constraint, i.e. only $\langle a \rangle$ is legal w.r.t. constraint C, the other four subsets of patterns can be pruned.

Step 3: construct $\langle a \rangle$-projected database and mine it. Only the sequences in SDB containing item a and satisfying constraint C should be projected. The $\langle a \rangle$-projected database, $SDB|_{\langle a \rangle} = \{\langle (_b)(bc)dd \rangle, \langle (_e)(abc)dd \rangle, \langle ddcb \rangle\}$. Notice that $\langle e(ab)(bc)dd \rangle$ is projected as $\langle (_b)(bc)dd \rangle$, where symbol "_" in the first transaction indicates that it is in the same transaction with a.

During the construction of the $\langle a \rangle$-projected database, we also find locally frequent items: (1) b can be inserted into the same transaction with a to form a longer frequent prefix $\langle (ab) \rangle$, and (2) $\langle b \rangle$, $\langle c \rangle$ and $\langle d \rangle$ can be concatenated to $\langle a \rangle$ to form longer frequent prefixes, i.e., $\langle ab \rangle$, $\langle ac \rangle$ and $\langle ad \rangle$. Locally infrequent items, such as e, should be ignored in the remaining mining of this projected database.

Then the set of patterns with prefix $\langle a \rangle$ can be further divided into five subsets without overlap: (1) pattern $\langle a \rangle$ itself; (2) those with prefix $\langle (ab) \rangle$; (3) those with prefix $\langle ab \rangle$; (4) those with prefix $\langle ac \rangle$; and (5) those with prefix $\langle ad \rangle$. By examining constraint C, one can see that pattern $\langle a \rangle$ fails C and thus is discarded; and $\langle (ab) \rangle$ is illegal w.r.t. constraint C, so the second subset of patterns is pruned. The remaining subsets of patterns should be explored one by one.

Step 4: recursive mining. The mining proceeds recursively. To mine patterns having $\langle ab \rangle$ as a prefix, we form the $\langle ab \rangle$-projected database $TDB|_{\langle ab \rangle} = \{\langle (_c)dd \rangle, \langle (_c)dd \}$. By recursively mining the projected database, we identify sequential pattern $\langle a(bc)d \rangle$ which satisfies the constraint.

To mine patterns with prefix $\langle ac \rangle$, we form $\langle ac \rangle$-projected database $TDB|_{\langle ac \rangle} = \{\langle dd \rangle, \langle dd \rangle, \langle b \rangle$. Every sequence in the projected database contains no sub-sequence satisfying the constrain. Thus, the search within $TDB|_{\langle ac \rangle}$ can be pruned. In other words, we will never search any projected database which does not potentially support patterns satisfying the constraint.

Similarly, we search the $\langle ad \rangle$-projected database and find $\langle add \rangle$ is a sequential pattern satisfying the constraint.

In summary, during the recursive mining, if the prefix itself is a pattern satisfying the constraint, it should be an output. The prefixes legal w.r.t. the constraint should be grown and mined recursively. The process terminates when there is no local frequent item or there is no legal prefix. It results in two final patterns: $\{\langle a(bc)d \rangle, \langle add \rangle\}$.

The correctness and completeness of the mining process in the above example can be verified. Limited by space, we omit the details here. The constrained sequential pattern mining algorithm is outlined as follows.

ALGORITHM 1. (prefix-growth) *Mining sequential patterns with prefix-monotone constraints.*

Input: *A sequence database SDB, support threshold min_sup, and prefix-monotone constraint C;*

Output: *The complete set of sequential patterns satisfying C;*

Method:

call $prefix_growth(\langle\rangle, SDB)$.

Function $prefix_growth(\alpha, SDB|_\alpha)$

// α: prefix; $SDB|_\alpha$: the α-projected database

Step 1. *Let l be the length of α. Scan $SDB|_\alpha$ once, find length-$(l+1)$ frequent prefix in $SDB|_\alpha$, and remove infrequent items and useless sequences;*

Step 2. for each *length-$(l+1)$ frequent prefix α' potentially satisfying the constraint C* do

Step 2a. if *α' satisfies C*, then *output α' as a pattern;*

Step 2b. *form $SDB|_{\alpha'}$;*

Step 2c. call $prefix_growth(\alpha', SDB|_{\alpha'})$ □

Is prefix-growth *an efficient algorithm?* First, Algorithm prefix-growth takes PrefixSpan as the basic sequential pattern mining algorithm and pushes prefix-monotone constraints deeply into the PrefixSpan mining process. The performance study in [9] shows that PrefixSpan outperforms GSP, owing to (1) PrefixSpan adopts a prefix growth and database projection framework: for each frequent prefix subsequence, only its corresponding suffix subsequences need to be projected and examined without candidate generation; (2) it applies a divide-and-conquer strategy so that sequential patterns are grown by exploring only local frequent patterns in each projected database; and (3) it explores further optimizations, including a *pseudo-projection* technique when the projected database and its associated pseudo-projection processing structure fits in main memory, etc.

Second, prefix-growth handles a broader scope of constraints than anti-monotonicity and monotonicity. A typical such example is regular expression constraints, which is difficult to be explored using an Apriori-based method, as shown in SPIRIT. By prefix-growth, such constraints can be naturally pushed deep into the mining process.

Interestingly, both prefix-growth and SPIRIT [3] push regular expression constraints by relaxing the constraint to achieve some nice property facilitating the constraint-based pruning. However, the SPIRIT methods requires that the relaxed constraints must have the strong anti-monotonic property, while prefix-growth only enforces the prefix-monotonic property which is weaker than the anti-monotonicity. As will be shown in the experimental results, prefix-growth outperforms SPIRIT in pushing regular expression constraints.

Third, constraint checking in Step 1 further shrinks projected databases effectively, due to its removal of useless sequences w.r.t. a given constraint during the prefix growth.

People may wonder whether Apriori-based methods, such as GSP and SPADE, can do similar prefix-based pruning using prefix-monotone constraints. Taking a non-anti-monotonic regular expression constraint as an example, for Apriori-based methods, a pattern whose prefix failing a constraint cannot be pruned since inserting more items/transactions to the pattern at *some other* positions may still lead to a valid pattern. However, by exploring prefix-monotone constraints, prefix-growth puts stronger restrictions on the possible subsequences to grow and thus prunes search space more effectively.

5. HANDLING TOUGH AGGREGATE CONSTRAINTS IN THE NEW FRAMEWORK

Besides regular expression constraints, one may wonder whether prefix-growth can handle effectively the two tough aggregate constraints in Table 2, average and g_sum? Both constraints are neither anti-monotonic nor monotonic. Even worse, they are not prefix-monotone!

For example, let us mine sequential patterns with constraint $C \equiv avg(\alpha) \leq 25$ in a sequence database SDB of Table 3, with support threshold = 2, and four items with values 10, 20, 30 and 50, respectively. For convenience, item value is used as its ID.

Sequence_id	Sequence
10	$\langle 50\ 10\ 20\ 20\rangle$
20	$\langle 30\ 50\ 20\rangle$
30	$\langle 50\ 10\ 20\ 10\ 10\rangle$
40	$\langle 30\ 20\ 10\rangle$

Table 3: Another sequence database SDB.

Constraint C cannot be pushed naively into the PrefixSpan mining process. For example $\alpha = \langle 50\rangle$ cannot be discarded even $avg(\alpha) \not\leq 25$, since by appending more elements to α, we may have $\alpha' = \langle 50\ 10\ 20\ 10\rangle$ and $avg(\alpha') \leq 25$. Also, one can easily verify C is not prefix-monotone.

In [8], a technique is developed to push *convertible* constraints, like $avg(X) \geq 25$, into frequent pattern mining on *transactional databases*. The general idea is to use a proper order of frequent items, like value descending order for constraint $avg(X) \geq v$, such that the list of frequent items according to the order has a nice anti-monotonic or monotonic property.

Can we apply the technique in [8] to attack aggregate constraints for sequential pattern mining? Unfortunately, the answer is negative. For every sequence, a temporal order has been pre-composed and we do not have the freedom to re-arrange the items in sequences. The trick of simple ordering does not work well here.

Value-ascending order over the set of items should be used to determine the order of projected databases to be processed. An item i is called a small item if its value $i.value \leq v$, otherwise, it is called a big item.

In the first scan of a (projected) database, unpromising big items in sequences should be removed according to the following two rules.

Unpromising sequence pruning rule. For a sequence α, let n be the number of instances of small items and s be the sum of them[3]. A big item x in α is unpromising and should be removed if $(s + x.value)/(n+1)$ violates the constraint.

Similarly, unpromising sequence pruning rule can be applied recursively in an α-projected database. For a projection $\gamma = \beta/\alpha$, let n be the number of instances of small items appearing in γ but not in α and s be the sum of them. A big item x in α is unpromising and should be removed if $(s + sum(\alpha) + x.value)/(n + \#items(\alpha) + 1)$ violates the constraint[4].

Unpromising pattern pruning rule. An item marking method can be developed to mark and further prune some unpromising items as follows. In the α-projected database[5], when a pattern β is found where the first item following α is a small item, we check whether that small item can be replaced by a big item x frequent in the projected database and still can get average value satisfying the constraint. If so, prefix $\langle \alpha \cdot x\rangle$ is marked *promising* and does not need to be checked and marked again in this projected database. When all patterns with some small item as the first one following α have been found, for the prefixes with a big item x following α having not been marked, $\langle \alpha \cdot x\rangle$ as well as the projected databases can be pruned if $\langle \alpha \cdot x\rangle$ violates the constraint.

The rationale of this rule is as follows. For a big item x, if $\langle \alpha \cdot x\rangle$ violates the constraint but $\langle \alpha \cdot x \cdot \beta\rangle$ is a sequential pattern satisfying the constraint, then there must be some $\beta' \sqsubseteq \beta$ such that β' starts with a small item and $\langle \alpha \cdot \beta'\rangle$ is a sequential pattern satisfying the constraint.

These rules can be proved easily. Limited by space, we illustrate them by mining SDB in Table 3 with constraint $C \equiv avg(\alpha) \leq 25$.

In the first scan of SDB, we remove the unpromising big items in sequences by applying the *unpromising sequence pruning rule*. For example, in the second sequence, 20 is the only small item and $\frac{20+50}{2} = 35 > 25$. This sequence cannot support andy sequential patterns having item 50 and satisfying the constraint C. Thus, item 50 in the second sequence should be moved.

In the same database scan, we also find length-1 patterns, $\langle 10\rangle$, $\langle 20\rangle$, $\langle 30\rangle$ and $\langle 50\rangle$. The set of patterns can be partitioned into four subsets without overlap: (1) those with prefix $\langle 10\rangle$; (2) those with prefix $\langle 20\rangle$; (3) those with prefix $\langle 30\rangle$; and (4) those with prefix $\langle 50\rangle$. These subsets of patterns should be explored one by one in this order.

Step 1. The set of patterns with prefix $\langle 10\rangle$ can be found by constructing $\langle 10\rangle$-projected database and mining it recursively. The items in $\langle 10\rangle$-projected database are small ones, so all patterns in it have average no greater than 25 and thus satisfy the constraint. There are two patterns there: $\langle 10\rangle$ and $\langle 10\ 20\rangle$.

When pattern $\langle 10\rangle$ is found, it can be regarded as a small item 10 following a prefix $\langle\rangle$. Thus, we apply the *unpromising pattern pruning rule* to mark and prune patterns. Prefix $\langle 30\rangle$ is marked as *promising*, since $avg(\langle 30\rangle \cdot \langle 10\rangle) = 20 < 25$. Prefix $\langle 30\rangle$ will not be checked against any other pattern after it is marked. None of the patterns with prefix $\langle 10\rangle$ can be used to mark prefix $\langle 50\rangle$.

Step 2. Similarly, we can find patterns with prefix $\langle 20\rangle$ by constructing and mining $\langle 20\rangle$-projected database. They are $\langle 20\rangle$ and $\langle 20\ 10\rangle$. None of the patterns with prefix $\langle 20\rangle$ can be used to mark prefix $\langle 50\rangle$.

Step 3. 30 is a big item and prefix $\langle 30\rangle$ violates the constraint. For patterns with prefix $\langle 30\rangle$, since prefix $\langle 30\rangle$ is marked, we need to construct $\langle 30\rangle$-projected database and mine it. Pattern $\langle 30\ 20\rangle$ is found.

Step 4. Prefix $\langle 50\rangle$ has not been marked. According to the unpromising pattern pruning rule, no pattern with prefix $\langle 50\rangle$ can satisfy the constraint. We do not need to construct or mine $\langle 50\rangle$-projected database.

With the same spirit, constraints $avg(\alpha) \geq v$ and $sum(\alpha)\ \theta\ v$ (where $\theta \in \{\leq, \geq\}$, and items can be with non-negative and negative values) can also be pushed deep into prefix-growth mining process.

In summary, with minor revision, prefix-growth can be extended to handle some tough aggregate constraints without prefix-monotone property. With such extensions, all established advantages of prefix-growth still retain and the pruning is still sharp.

6. EXPERIMENTAL RESULTS AND PERFORMANCE STUDY

To evaluate the effectiveness and efficiency of the algorithms, we performed an extensive experimental evaluation on both synthetic and real datasets. The results are consistent. Limited by space, in this section, we report only the results on some synthetic datasets generated by the IBM data generator described in [2].

[3] If there are multiple instances of one small item, the value of that item should be counted multiple times.

[4] Function $\#items(\alpha)$ returns the number of instances of items in sequence α.

[5] The whole database SDB can be regarded as $SDB|_{\langle\rangle}$.

All the experiments are performed on a 700MHz AMD PC machine with 256 megabytes main memory, running Microsoft Windows 2000 Server. All methods are implemented using Microsoft Visual C++ 6.0. We compare performance of four methods: (1) GSP as described in [1]. We also revised GSP to push anti-monotonic and monotonic constraints. (2) SPIRIT as described in [3]. We implemented SPIRIT(V), the overall fastest one among the SPIRIT family. (3) SPADE [11]. We got the source code from the author. We only study the performance of SPADE on mining *without constraint*. Revision of SPADE to handle constraints is non-trivial. (4) Prefix-growth, the algorithm developed in this paper. We adopted the level-by-level projection and pseudo-projection techniques described in [9].

We first compare the efficiency of mining sequential patterns without constraint. Figure 1 shows the scalability of prefix-growth, GSP and SPADE with support threshold on dataset $C10T5S4I1.25D200k$, which contains $100,000$ sequences with $10,000$ items. The expected average number of items within a transaction is 5 (denoted as $T5$ and the expected average number of transaction in maximal sequential pattern is 4 (denoted as $S4$).

As can be seen from the figure, prefix-growth is more efficient and scalable than GSP and SPADE, while SPADE is faster than GSP, especially when support threshold is low. This comparison confirms the inherent advantage of prefix-growth over GSP and SPADE. In the remaining experiments, we study whether prefix-growth can carry the advantage to the extent of mining with constraints.

To evaluate the effect of a constraint on mining sequential patterns, we define the **selectivity** of a constraint as the ratio of the number of patterns FAILING the constraint against the total number of patterns. Therefore, a constraint with 0% selectivity filters out no pattern, while one with 100% selectivity filters out all the patterns.

Anti-monotonic constraints can be pushed deep into GSP. We modified GSP such that it only generates candidates satisfying the constraint. Thus, both GSP and prefix-growth can push anti-monotonic constraints deep into the mining processes. To show the capability of GSP and prefix-growth in pushing anti-monotonic constraints into mining, we use constraint $Dur(\alpha) \leq \Delta t$ as an example here. With various values of Δt, the constraint achieves various selectivity. The support threshold is fixed to 0.7%. For GSP and prefix-growth, we compare the runtime with constraint to the one without constraint, respectively, and plot Figure 2. The relative runtime is the ratio of the runtime of an algorithm with constraint over its runtime without constraint. In this way, the effect of constraint pushing on runtime improvement can be measured objectively. In general, both methods are capable in pushing anti-monotone constraints. When the constraint selectivity is weak, since most patterns have to be generated and tested, not too much time can be saved. However, when the selectivity is high, i.e., many patterns do not satisfy the constraint, a major saving can be observed and prefix-growth performs better. Comparing constraint pushing in GSP and prefix-growth, prefix-growth uses the constraint to prune both the patterns and the sequences in projected databases, while GSP has to search from the whole database all the time. When mining in large databases, the database search cost in GSP is non-trivial.

Monotonic constraints can be used to save the cost of constraint checking, but it cannot cut the search space. In our experiments, since we use relatively simple constraints, such as $Dur(\alpha) \geq \Delta t$, the cost of constraint checking is CPU-bound. However, the cost of the whole mining process is I/O-bound. This makes the effect of pushing monotonic constraint into the mining process hard to be observed from runtime reduction. However, if we look at the number of constraint tests performed, the advantage of monotonic constraint pushing can be evaluated objectively. /nopBy pushing a monotonic constraint, prefix-growth can save a lot of effort on constraint testing. Therefore, in the experiment about pushing monotonic constraint, the number of constraint tests is used as the performance measure. We also revise GSP to handle monotonic constraints. Once a monotonic constraint is satisfied by a pattern, all candidates which are supersets of this pattern do not need to be checked anymore. Our results show that GSP and prefix-growth follow a similar trend on saving of constraint checking: the higher the constraint selectivity, the more saving. Prefix-growth performs better. Limited by space, we omit the details here.

The complexity of regular expression constraints can be roughly measured by the number of state changes (i.e., edges) in their corresponding deterministic finite automata. For each level of complexity, we randomly generate $1,000$ constraints and test both SPIRIT(V) and prefix-growth on them. The support threshold is set to 0.2%. The results are shown in Table 4. With simple regular expression constraints, both SPIRIT(V) and prefix-growth are efficient. SPIRIT(V) is even better when the expression contains only two state changes. However, when the complexity of the constraints goes up, the average runtime of SPIRIT(V) increases dramatically. The increase of average runtime of prefix-growth is much more moderate. Even with rather complicated constraints, prefix-growth is still very efficient. The results show that prefix-growth is more scalable and efficient than SPIRIT(V) in pushing regular expression constraints.

Number of state changes	Average runtime of SPIRIT(V)	Average runtime of prefix-growth
10	199.176	1.20
9	98.241	1.00
8	48.540	0.89
7	23.824	0.82
6	11.500	0.71
5	5.400	0.67
4	2.453	0.61
3	1.031	0.60
2	0.381	0.57

Table 4: Experimental results on mining with regular expression constraints (runtime is measured in seconds.

Based on our analysis, the difference between the two methods in performance can be explained as follows. With regular expression constraints, prefix-growth can prune both patterns and projected databases. However, SPIRIT(V) has to scan the whole sequence database repeatedly. On the other hand, even when SPIRIT(V) has pruned many candidates, it still generates some candidates and has to test them against the whole database.

We also test prefix-growth on pushing constraint $avg(\alpha) \leq v$ to sequential pattern mining. The support threshold is set to 0.2%. The result is shown in Figure 3. As can be seen, prefix-growth is efficient and scalable w.r.t. selectivity of the constraint. To test the effect of prefix marking and pruning technique in prefix-growth for mining with

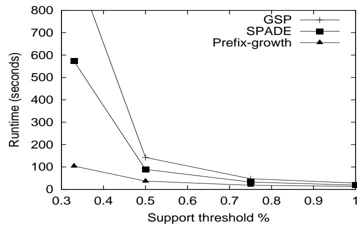

Figure 1: Scalability of GSP, SPADE and prefix-growth without constraint.

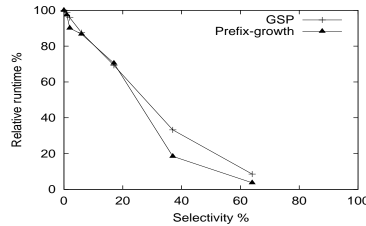

Figure 2: Capability of GSP and prefix-growth on pushing anti-monotone constraint.

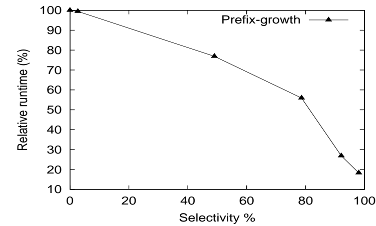

Figure 3: Scalability of prefix-growth with constraint $avg(\alpha) \leq v$.

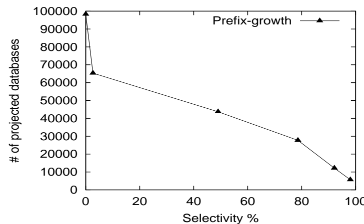

Figure 4: Number of projected database in prefix-growth with constraint $avg(\alpha) \leq v$.

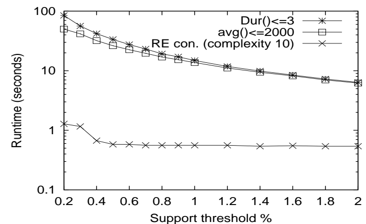

Figure 5: Scalability of prefix-growth w.r.t. support threshold.

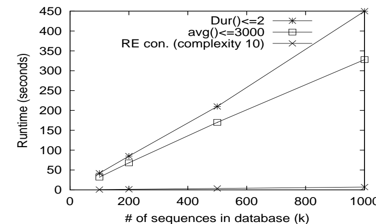

Figure 6: Scalability of prefix-growth w.r.t. database size.

constraint $avg(\alpha) \leq v$, we count the number of projected databases in Figure 4. As can be seen, the prefix marking technique in prefix-growth prunes a good number of projected databases and contributes substantially to the scalability of prefix-growth. It is also interesting to see that the curves in Figure 3 and 4 share similar shape. This indicates that the major cost in prefix-growth is mining projected databases. As the number of projected databases can be cut, the runtime can be brought down accordingly.

We tested the scalability of pushing various constraints in prefix-growth w.r.t. support threshold. The results are shown in Figure 5. We also test the scalability of prefix-growth w.r.t. database size when mining with various constraints. The results are shown in Figure 6.

In summary, the experimental results and performance study show that prefix-growth is efficient and scalable in mining sequential patterns with various constraints. That strongly supports our theoretical analysis.

7. CONCLUSIONS

In this paper, we have systematically studied the problem of pushing various constraints deep into sequential pattern mining. We characterize constraints for sequential pattern mining from both the application and constraint-pushing points of views. A general property of constraints for sequential pattern mining, prefix-monotone property, is identified. It covers many commonly used constraints. An efficient algorithm, prefix-growth, is developed to push prefix-monotone constraints deep into the mining process. With some minor extensions, some tough constraints, like those involving aggregate $avg()$ and $sum()$, can also be pushed deep into prefix-growth. Our extensive experimental results and performance study show that prefix-growth is efficient and scalable in mining large databases.

We have been working on a systematic implementation of constraint-based sequential pattern mining in a data mining system. Prefix-growth represents a new and promising methodology at effective and efficient mining sequential patterns with constraints. It is interesting to extend it towards mining sequential patterns with other more complicated constraints, and mining other kinds of time-related knowledge with various constraints.

ACKNOWLEDGEMENTS. The work was supported in part by research grants from NSERC and NCE of Canada, and the University of Illinois, and a gift from Microsoft Research. We thank Dr. Mohammed J. Zaki for providing us the source code of SPADE.

8. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. *VLDB'94*.
[2] R. Agrawal and R. Srikant. Mining sequential patterns. *ICDE'95*.
[3] M. Garofalakis, R. Rastogi, and K. Shim. Spirit: Sequential pattern mining with regular expression constraints. *VLDB'99*.
[4] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. *SIGMOD'00*.
[5] H. Mannila, H Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1:259–289, 1997.
[6] R. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. *SIGMOD'98*.
[7] J. Pei and J. Han. Can we push more constraints into frequent pattern mining? *KDD'00*.
[8] J. Pei et al. Mining frequent itemsets with convertible constraints. *ICDE'01*.
[9] J. Pei et al. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. *ICDE'01*.
[10] R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. *SIGMOD'96*.
[11] M. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning*, 40:31–60, 2001.