Pattern-growth Methods for Sequential Pattern Mining: Principles and Extensions *

Jiawei Han
Department of Computer Science
University of Illinois at Urbana-Champaign
School of Computing Science
Simon Fraser University
Email: han@cs.sfu.ca

Jian Pei
School of Computing Science
Simon Fraser University
Email: peijian@cs.sfu.ca

Abstract

Sequential pattern mining is an important data mining problem with broad applications. It is challenging since one may need to examine a combinatorially explosive number of possible subsequence patterns. Most of the previously developed sequential pattern mining methods follow the methodology of Apriori which may substantially reduce the number of combinations to be examined. However, Apriori-like methods still encounter problems when a sequence database is large and/or when sequential patterns to be mined are numerous and/or long.

Recently, we proposed two pattern-growth methods, FreeSpan [6] and PrefixSpan [11], for efficient sequential pattern mining. These methods explore efficient database projections guided by patterns already found. Performance studies show that both methods outperform Apriori-based GSP method and PrefixSpan achieves the best performance in mining large sequence databases.

In this paper, we provide an overview of pattern-growth sequential pattern mining methods. Moreover, pattern-growth methods can be extended to mine many other kinds of temporal patterns from large time-related databases. As an example, we show that with minor modification, PrefixSpan can be extended to mine sequential patterns with regular expression constraints, which is an interesting mining problem proposed in [4].

1 Introduction

Sequential pattern mining, which discovers frequent subsequences as patterns in a sequence database, is an important data mining problem with broad applications, including the analyses of customer purchase behavior, Web access patterns, scientific experiments, disease treatments, natural disasters, DNA sequences, and so on.

The sequential pattern mining problem was first introduced by Agrawal and Srikant in [2]: *Given a set of sequences, where each sequence consists of a list of elements and each element consists of a set of items, and given a user-specified min_support threshold, sequential pattern mining is to find all of the frequent subsequences, i.e., the subsequences whose occurrence frequency in the set of sequences is no less than min_support.*

Many studies have contributed to the efficient mining of sequential patterns or other frequent patterns in time-related data, e.g., [2, 12, 9, 10, 3, 8, 5, 4]. Almost all of the previously proposed methods for mining sequential patterns and other time-related frequent patterns are Apriori-like, i.e., based on the Apriori property proposed in association mining [1], which states the fact that *any super-pattern of a nonfrequent pattern cannot be frequent.*

Based on this heuristic, a typical Apriori-like method such as GSP [12] adopts a multiple-pass, candidate-generation-and-test approach in sequential pattern mining. This is outlined as follows. The first scan finds all of the frequent items which form the set of single item frequent sequences. Each subsequent pass starts with a *seed set* of sequential patterns, which is the set of sequential patterns found in the previous pass. This seed set is used to generate new potential patterns, called *candidate sequences*. Each candidate sequence contains one more item than a seed sequential pattern, where each element in the pattern may contain one or multiple items. The number of items in a sequence is called the *length* of the sequence. So, all the candidate sequences in a pass will have the same length. The scan of the database in one pass finds the support for each candidate sequence. All of the candidates whose support in the database is no less than min_support form the set of the newly found sequential patterns. This set then becomes the seed set for the next pass. The algorithm terminates when no new sequential pattern is found in a pass, or no candidate sequence can be generated.

Similar to the analysis of Apriori frequent pattern mining method in [7], one can observe that the Apriori-like sequential pattern mining method, though reduces search space, bears three nontrivial, inherent costs which are independent of detailed implementation techniques.

*The work was supported in part by grants from the Natural Sciences and Engineering Research Council of Canada, the Networks of Centres of Excellence of Canada (grant NCE/IRIS-3), and from the Hewlett-Packard Lab., U.S.A.

- **Potentially huge set of candidate sequences.** Since the set of candidate sequences includes all the possible permutations of the elements and repetition of items in a sequence, an Apriori-based method may generate a really large set of candidate sequences even for a moderate seed set. For example, if there are 1000 frequent sequences of length-1, such as $\langle a_1 \rangle$, $\langle a_2 \rangle$, ..., $\langle a_{1000} \rangle$, an Apriori-like algorithm will generate $1000 \times 1000 + \frac{1000 \times 999}{2} = 1,499,500$ candidate sequences, where the first term is derived from the set $\langle a_1 a_1 \rangle$, $\langle a_1 a_2 \rangle$, ..., $\langle a_1 a_{1000} \rangle$, $\langle a_2 a_1 \rangle$, $\langle a_2 a_2 \rangle$, ..., $\langle a_{1000} a_{1000} \rangle$, and the second term is derived from the set $\langle (a_1 a_2) \rangle$, $\langle (a_1 a_3) \rangle$, ..., $\langle (a_{999} a_{1000}) \rangle$.

- **Multiple scans of databases.** Since the length of each candidate sequence grows by one at each database scan, to find a sequential pattern $\{(abc)(abc)(abc)(abc)(abc)\}$, an Apriori-based method must scan the database at least 15 times.

- **Difficulties at mining long sequential patterns.** A long sequential pattern must grow from a combination of short ones, but the number of such candidate sequences is exponential to the length of the sequential patterns to be mined. For example, suppose there is only a single sequence of length 100, $\langle a_1 a_2 \ldots a_{100} \rangle$, in the database, and the min_support threshold is 1 (i.e., every occurring pattern is frequent), to (re-)derive this length-100 sequential pattern, the Apriori-based method has to generate 100 length-1 candidate sequences, $100 \times 100 + \frac{100 \times 99}{2} = 14,950$ length-2 candidate sequences, $\left(\begin{array}{c} 100 \\ 3 \end{array} \right) = 161,700$ length-3 candidate sequences[1], ... Obviously, the total number of candidate sequences to be generated is greater than $\Sigma_{i=1}^{100} \left(\begin{array}{c} 100 \\ i \end{array} \right) = 2^{100} - 1 \approx 10^{30}$.

In many applications, it is not unusual that one may encounter a large number of sequential patterns and long sequences, such as stock sequence analysis. Therefore, it is important to re-examine the sequential pattern mining problem to explore more efficient and scalable methods.

Based on our analysis, both the thrust and the bottleneck of an Apriori-based sequential pattern mining method come from its step-wise candidate sequence generation and test. Can we develop a method which may absorb the spirit of Apriori but avoid or substantially reduce the expensive candidate generation and test?

[1]Notice that Apriori does cut a substantial amount of search space. Otherwise, the number of length-3 candidate sequences would have been $100 \times 100 \times 100 + 100 \times 100 \times 99 + \frac{100 \times 99 \times 98}{3 \times 2} = 2,151,700$.

With this motivation, recently, the pattern-growth methodology [7] has been extended to sequential pattern mining and two methods, FreeSpan [6] and PrefixSpan [11], are proposed. Performance studies show that they outpeform Apriori-like method GSP and PrefixSpan achieves best performance in mining large sequence databases.

Moreover, many applications look for many other kinds of temporal patterns from large time-related databases. *Can pattern-growth methods be extended to attack those temporal pattern mining problems effectively and efficiently?*

As an example, let us consider mining sequential patterns with regular expression constraints, as proposed in [4]. For instance, to help planning advertisement placement, an automobile business analyst may want to find sequential patterns in user web access logs such that patterns contain at least one web page of car manufacturer and at least 2 sites about traveling. Such a mining query can be expressed effectively using a regular expression. Efficient mining sequential patterns with regular expressions helps people locate the to-the-point knowledge effectively.

In this paper, we provide an overview of pattern-growth sequential pattern mining methods. Moreover, as an example of mining other kinds of temporal patterns using pattern-growth methods, we discuss an extension of pattern-growth methods to mine sequential patterns with regular expression. Our study indicates that pattern-growth methods are effectively, efficient, and scalable for mining large sequence databases. The pattern-growth methodology opens a door towards effective and efficient mining many other kinds of temporal patterns from large time-related databases.

The remaining of the paper is organized as follows. In Section 2, we revisit the sequential pattern mining problem and the two pattern-growth mining methods, FreeSpan and PrefixSpan. An empirical evaluation on some sequential pattern mining methods is presented in Section 3. In Section 4, we consider mining sequential patterns with regular expression constraints. The paper is concluded in Section 5.

2 Sequential Pattern Mining and Pattern-growth Methods

In this section, we first define the problem of sequential pattern mining, and then illustrate an Apriori-like method, GSP, and two recently proposed pattern-growth methods, FreeSpan and PrefixSpan, using examples.

2.1 Problem Definition and GSP

Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of all **items**. An **itemset** is a subset of items. A **sequence** is an ordered list of itemsets. A sequence s is denoted by $\langle s_1 s_2 \cdots s_l \rangle$, where s_j is an itemset, i.e., $s_j \subseteq I$ for $1 \leq j \leq l$. s_j is also called an **element** of the sequence, and denoted as $(x_1 x_2 \cdots x_m)$, where x_k is an item, i.e., $x_k \in I$ for $1 \leq k \leq m$. For brevity, the brackets are omitted if an element has only one item. That is, element (x) is written as x. An item can occur at most once in an element of a sequence, but can occur multiple times in different elements of a sequence. The number of instances of items in a sequence is called the **length** of the sequence. A sequence with length l is called an **l-sequence**. A sequence $\alpha = \langle a_1 a_2 \cdots a_n \rangle$ is called a **subsequence** of another sequence $\beta = \langle b_1 b_2 \cdots b_m \rangle$ and β a **super sequence** of α, denoted as $\alpha \sqsubseteq \beta$, if there exist integers $1 \leq j_1 < j_2 < \cdots < j_n \leq m$ such that $a_1 \subseteq b_{j_1}$, $a_2 \subseteq b_{j_2}$, ..., $a_n \subseteq b_{j_n}$.

A **sequence database** S is a set of tuples $\langle sid, s \rangle$, where sid is a **sequence_id** and s is a sequence. A tuple $\langle sid, s \rangle$ is said to *contain* a sequence α, if α is a subsequence of s, i.e., $\alpha \sqsubseteq s$. The support of a sequence α in a sequence database S is the number of tuples in the database containing α, i.e., $support_S(\alpha) = |\ \{\langle sid, s \rangle | (\langle sid, s \rangle \in S) \wedge (\alpha \sqsubseteq s)\}\ |$. It can be denoted as $support(\alpha)$ if the sequence database is clear from the context. Given a positive integer ξ as the **support threshold**, a sequence α is called a (frequent) **sequential pattern** in sequence database S if the sequence is contained by at least ξ tuples in the database, i.e., $support_S(\alpha) \geq \xi$. A sequential pattern with length l is called an **l-pattern**.

Example 1 Let our running database be *sequence database* S given in Table 1 and *min_support* $= 2$. The set of *items* in the database is $\{a, b, c, d, e, f, g\}$.

Sequence_id	Sequence
10	$\langle a(abc)(ac)d(cf) \rangle$
20	$\langle (ad)c(bc)(ae) \rangle$
30	$\langle (ef)(ab)(df)cb \rangle$
40	$\langle eg(af)cbc \rangle$

Table 1: A sequence database

A *sequence* $\langle a(abc)(ac)d(cf) \rangle$ has five *elements*: (a), (abc), (ac), (d) and (cf), where items a and c appear more than once respectively in different elements. It is also a 9-*sequence* since there are 9 instances appearing in that sequence. Item a happens three times in this sequence, so it contributes 3 to the *length* of the sequence. However, the whole sequence $\langle a(abc)(ac)d(cf) \rangle$ contributes only one to the *support* of $\langle a \rangle$. Also, sequence $\langle a(bc)df \rangle$ is a *subsequence* of $\langle a(abc)(ac)d(cf) \rangle$. Since both sequences 10 and 30 *contain* subsequence $s = \langle (ab)c \rangle$, s is a *sequential pattern* of length 3 (i.e., 3-*pattern*). $\square$

Problem Statement. Given a sequence database and a *min_support* threshold, the problem of **sequential pattern mining** is to find the complete set of sequential patterns in the database.

Apriori heuristic, which is an anti-monotone property, holds for sequential patterns: *every non-empty subsequence of a sequential pattern is a sequential pattern.* With the Apriori heuristic, a typical sequential pattern mining method, GSP [12], proceeds as shown in the following example.

Example 2 (GSP) Given the database S and min_support in Example 1, GSP first scans S, collects the support for each item, and finds the set of frequent items (in the form of *item* : *support*) as below,

$$a:4, b:4, c:4, d:3, e:3, f:3, g:1$$

By filtering infrequent items, g, we obtain the first seed set $L_1 = \{\langle a \rangle, \langle b \rangle, \langle c \rangle, \langle d \rangle, \langle e \rangle, \langle f \rangle\}$, each representing a 1-element sequential pattern. Each subsequent pass starts with the seed set found in the previous pass and uses it to generate new potential sequential patterns, called *candidate sequences.*

For L_1, a set of 6 length-1 sequential patterns generates a set of $6 \times 6 + \frac{6 \times 5}{2} = 51$ candidate sequences, $C_2 = \{\langle aa \rangle, \langle ab \rangle, \ldots, \langle af \rangle, \langle ba \rangle, \langle bb \rangle, \ldots, \langle ff \rangle, \langle (ab) \rangle, \langle (ac) \rangle, \ldots, \langle (ef) \rangle\}$.

The multi-scan mining process is shown in Figure 1, with the following explanations.

The set of candidates is generated by a self-join of the sequential patterns found in the previous pass. In the k-th pass, a sequence is a candidate only if each of its length-$(k-1)$ subsequences is sequential patterns found at the $(k-1)$-st pass.

A new scan of the database collects the support for each candidate sequence and finds the new set of sequential patterns. This set becomes the seed for the next pass. The algorithm terminates when no sequential pattern is found in a pass, or when there is no candidate sequence generated.

The number of scans is at least the maximum length of sequential patterns. It needs one more scan if the sequential patterns obtained in the last scan still generate new candidates.

GSP, though benefits from the Apriori pruning, still generates a large number of candidates. In this example, 6 length-1 sequential patterns generate 51 length-2 candidates, 22 length-2 sequential patterns generate 64 length-3 candidates, etc.

Candidates generated by GSP may not appear in the database at all. For example, 13 out of 64 length-3 candidates do not appear in the database. $\square$

2.2 Pattern-growth Methods

As analyzed before, the bottleneck of sequential pattern mining is candidate generation and test. To overcome

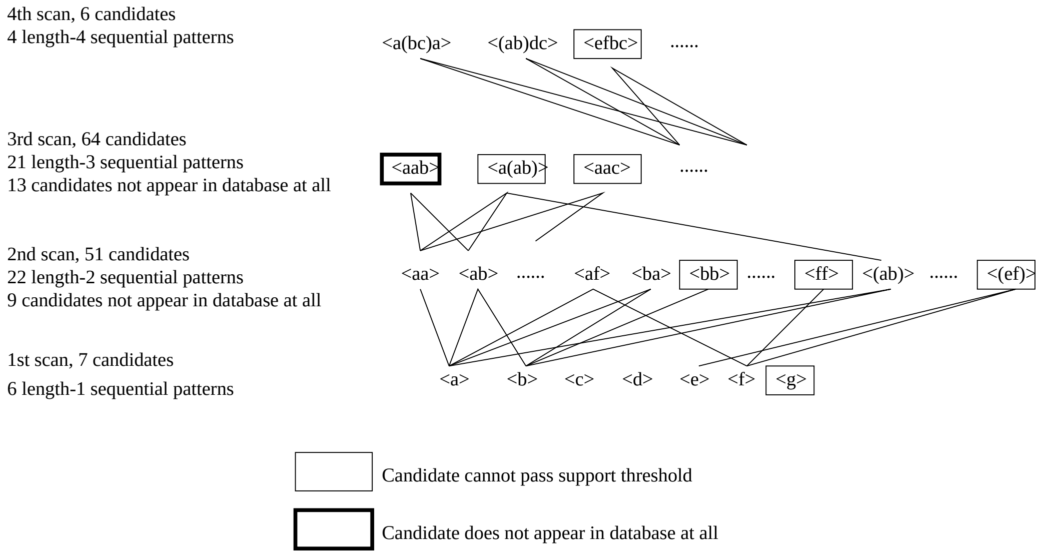
4th scan, 6 candidates
4 length-4 sequential patterns
<a(bc)a>
<(ab)dc>
<efbc>
......
3rd scan, 64 candidates
21 length-3 sequential patterns
13 candidates not appear in database at all
<aab>
<a(ab)>
<aac>
......
2nd scan, 51 candidates
22 length-2 sequential patterns
9 candidates not appear in database at all
<aa>
<ab>
......
<af>
<ba>
<bb>
......
<ff>
<(ab)>
......
<(ef)>
1st scan, 7 candidates
6 length-1 sequential patterns
<a>
<b>
<c>
<d>
<e>
<f>
<g>
Candidate cannot pass support threshold
Candidate does not appear in database at all

Figure 1: Candidates and sequential patterns in GSP

this difficulty, a new category of methods for sequential pattern mining, called *pattern-growth methods*, are developed in [6, 11]. They adopt a divide-and-conquer methodology and mine sequential patterns (almost) without candidate generation. These approaches have several distinct features:

1. Instead of generating a large number of candidates, the methods preserve (in some compressed forms) the essential groupings of the original data elements for mining. Then the analysis is focused on counting the frequency of the relevant data sets instead of candidate sets.
2. Instead of scanning the *entire* database to match against the *whole* corresponding set of candidates in each pass, the methods partition the data set to be examined as well as the set of patterns to be examined by database projection. Such a divide-and-conquer methodology substantially reduces the search space and leads to high performance.
3. With the growing capacity of main memory and the substantial reduction of database size by database projection as well as the space needed for manipulating large sets of candidates, a substantial portion of data can be put into main memory for mining. Pseudo-projection has been developed for pointer-based traversal. The performance studies have shown the effectiveness of such techniques.

Here, we illustrate the general ideas of FreeSpan and PrefixSpan using examples, respectively.

2.2.1 FreeSpan

To improve the performance of sequential pattern mining, a FreeSpan algorithm is developed in a recent study [6]. Its major ideas are illustrated in the following example.

Example 3 (FreeSpan) Given the database S and min_support in Example 1, FreeSpan first scans S, collects the support for each item, and finds the set of frequent items. Frequent items are listed in support descending order (in the form of *item : support*) as below,

$$\text{f\_list} = a:4, b:4, c:4, d:3, e:3, f:3$$

According to f_list, the complete set of sequential patterns in S can be divided into 6 disjoint subsets: (1) the ones containing only item a, (2) the ones containing item b but containing no items after b in f_list, (3) the ones containing item c but no items after c in f_list, and so on, and finally, (6) the ones containing item f.

The subsets of sequential patterns can be mined by constructing *projected databases*. Infrequent items, such as g in this example, are removed from construction of projected databases. The mining process is detailed as follows.

- **Finding sequential patterns containing only item a.** By scanning sequence database once, the only two sequential patterns containing only item a, $\langle a \rangle$ and $\langle aa \rangle$, are found.
- **Finding sequential patterns containing item b but no item after b in f_list.** This can be achieved by constructing the $\{b\}$-*projected database*. For a sequence α in S containing item b, a subsequence α' is derived by removing from α all items after b in f_list. α' is inserted into $\{b\}$-projected database. Thus, $\{b\}$-projected database contains

four sequences: $\langle a(ab)a\rangle$, $\langle aba\rangle$, $\langle (ab)b\rangle$ and $\langle ab\rangle$. By scanning the projected database once more, all sequential patterns containing item b but no item after b in f_list are found. They are $\langle b\rangle$, $\langle ab\rangle$, $\langle ba\rangle$, $\langle (ab)\rangle$.

- **Finding other subsets of sequential patterns.** Other subsets of sequential patterns can be found similarly, by constructing corresponding projected databases and mining them recursively.

Note that $\{b\}$-, $\{c\}$-, ..., $\{f\}$-projected databases are constructed simultaneously during one scan of the original sequence database. All sequential patterns containing only item a are also found in this pass.

This process is performed recursively on projected-databases. Since FreeSpan projects a large sequence database recursively into a set of small projected sequence databases based on the currently mined frequent sets, the subsequent mining is confined to each projected database relevant to a smaller set of candidates. Thus, FreeSpan is more efficient than GSP. □

The major cost of FreeSpan is to deal with projected databases. If a pattern appears in each sequence of a database, its projected database does not shrink (except for the removal of some infrequent items). For example, the $\{f\}$-projected database in this example is the same as the original sequence database, except for the removal of infrequent item g. Moreover, since a length-k subsequence may grow at any position, the search for length-$(k+1)$ candidate sequence will need to check every possible combination, which is costly.

2.2.2 PrefixSpan

To further improve the performance of sequential pattern mining, another pattern-growth method, PrefixSpan, is develop in [11].

Example 4 (PrefixSpan) Let us re-consider mining sequential patterns in sequence database S, shown in Table 1, with the support threshold set to 2. PrefixSpan works as follows.

First, we find length-1 sequential patterns by scanning S once. This derives the set of frequent items in sequences, i.e., the set of length-1 sequential patterns: $\{(\langle a\rangle : 4), (\langle b\rangle : 4), (\langle c\rangle : 4), (\langle d\rangle : 3), (\langle e\rangle : 3)$, and $(\langle f\rangle : 3)\}$.

Then, the search space can be partitioned into the following six subsets according to the six prefixes: (1) the ones with prefix $\langle a\rangle$; ...; and (6) the ones with prefix $\langle f\rangle$. The subsets of sequential patterns can be mined by constructing corresponding *projected databases* and mine each recursively. The projected databases as well as sequential patterns found in them are listed in Table 2, and the mining process is explained as follows.

The sequential patterns with prefix $\langle a\rangle$ are mined in the (prefix) $\langle a\rangle$-projected database. It is the collection that contains only those subsequences prefixed with the first occurrence of $\langle a\rangle$. For example, in sequence $\langle (ef)(ab)(df)cb\rangle$, only the subsequence $\langle (\_b)(df)cb\rangle$ should count. Notice that $(\_b)$ means that the last element in the prefix, which is a, together with b, form one element (i.e., occurring together). Thus the *$\langle a\rangle$-projected database* consists of four postfix sequences: $\langle (abc)(ac)d(cf)\rangle$, $\langle (\_d)c(bc)(ae)\rangle$, $\langle (\_b)(df)cb\rangle$ and $\langle (\_f)cbc\rangle$. By scanning $\langle a\rangle$-projected database once, all the length-2 sequential patterns prefixed with $\langle a\rangle$ can be found. They are: $(\langle aa\rangle : 2)$, $(\langle ab\rangle : 4)$, $(\langle (ab)\rangle : 2)$, $(\langle ac\rangle : 4)$, $(\langle ad\rangle : 2)$, and $(\langle af\rangle : 2)$.

Recursively, all sequential patterns with prefix $\langle a\rangle$ can be partitioned into 6 subsets: (1) that prefixed with $\langle aa\rangle$, (2) that with $\langle ab\rangle$, ..., and finally, (6) that with $\langle af\rangle$. These subsets can be mined by constructing respective projected databases and mining each recursively.

For example, the $\langle aa\rangle$-projected database consists of only one non-empty (postfix) subsequences prefixed with $\langle aa\rangle$: $\langle (\_bc)(ac)d(cf)\rangle$. Since there is no hope to generate any frequent subsequence from a single sequence, the processing of $\langle aa\rangle$-projected database terminates. Similarly, the $\langle ab\rangle$-projected database consists of three postfix sequences: $\langle (\_c)(ac)d(cf)\rangle$, $\langle (\_c)a\rangle$, and $\langle c\rangle$. Recursively mining it returns four sequential patterns: $\langle (\_c)\rangle$, $\langle (\_c)a\rangle$, $\langle a\rangle$, and $\langle c\rangle$ (i.e., $\langle a(bc)\rangle$, $\langle a(bc)a\rangle$, $\langle aba\rangle$, and $\langle abc\rangle$.)

Using the same method, sequential patterns with prefix $\langle b\rangle$, $\langle c\rangle$, $\langle d\rangle$, $\langle e\rangle$ and $\langle f\rangle$, can be mined from the corresponding projected databases respectively. The projected databases as well as the sequential patterns found are shown in Table 2. □

The example shows that PrefixSpan examines only the prefix subsequences and projects only their corresponding postfix subsequences into projected databases, and in each projected database, sequential patterns are grown by exploring only local frequent patterns.

To further improve mining efficiency, two kinds of optimizations are explored [11]: (1) *pseudo-projection*, and (2) *bi-level projection*. Pseudo-projection is based on the following idea: When the database can be held in main memory, instead of constructing a *physical* projection by collecting all the postfixes, one can use pointers referring to the sequences in the database as a *pseudo-projection*. Every projection consists of two pieces of information: *pointer* to the sequence in database and *offset* of the postfix in the sequence. This avoids physically copying postfixes. Thus, it is efficient in terms of both running time and space. However, it is not efficient if the pseudo-projection is used for disk-based accessing since random access of disk space is very costly. Therefore, when the sequence database cannot

Prefix	Projected (postfix) database	Sequential patterns
$\langle a\rangle$	$\langle(abc)(ac)d(cf)\rangle$, $\langle(\_d)c(bc)(ae)\rangle$, $\langle(\_b)(df)cb\rangle$, $\langle(\_f)cbc\rangle$	$\langle a\rangle$, $\langle aa\rangle$, $\langle ab\rangle$, $\langle a(bc)\rangle$, $\langle a(bc)a\rangle$, $\langle aba\rangle$, $\langle abc\rangle$, $\langle(ab)\rangle$, $\langle(ab)c\rangle$, $\langle(ab)d\rangle$, $\langle(ab)f\rangle$, $\langle(ab)dc\rangle$, $\langle ac\rangle$, $\langle aca\rangle$, $\langle acb\rangle$, $\langle acc\rangle$, $\langle ad\rangle$, $\langle adc\rangle$, $\langle af\rangle$
$\langle b\rangle$	$\langle(\_c)(ac)d(cf)\rangle$, $\langle(\_c)(ae)\rangle$, $\langle(df)cb\rangle$, $\langle c\rangle$	$\langle b\rangle$, $\langle ba\rangle$, $\langle bc\rangle$, $\langle(bc)\rangle$, $\langle(bc)a\rangle$, $\langle bd\rangle$, $\langle bdc\rangle$, $\langle bf\rangle$
$\langle c\rangle$	$\langle(ac)d(cf)\rangle$, $\langle(bc)(ae)\rangle$, $\langle b\rangle$, $\langle bc\rangle$	$\langle c\rangle$, $\langle ca\rangle$, $\langle cb\rangle$, $\langle cc\rangle$
$\langle d\rangle$	$\langle(cf)\rangle$, $\langle c(bc)(ae)\rangle$, $\langle(\_f)cb\rangle$	$\langle d\rangle$, $\langle db\rangle$, $\langle dc\rangle$, $\langle dcb\rangle$
$\langle e\rangle$	$\langle(\_f)(ab)(df)cb\rangle$, $\langle(af)cbc\rangle$	$\langle e\rangle$, $\langle ea\rangle$, $\langle eab\rangle$, $\langle eac\rangle$, $\langle eacb\rangle$, $\langle eb\rangle$, $\langle ebc\rangle$, $\langle ec\rangle$, $\langle ecb\rangle$, $\langle ef\rangle$, $\langle efb\rangle$, $\langle efc\rangle$, $\langle efcb\rangle$.
$\langle f\rangle$	$\langle(ab)(df)cb\rangle$, $\langle cbc\rangle$	$\langle f\rangle$, $\langle fb\rangle$, $\langle fbc\rangle$, $\langle fc\rangle$, $\langle fcb\rangle$

Table 2: Projected databases and sequential patterns

be held in main memory, a *bi-level projection* method is explored, which projects databases not at every level but at every two levels. In comparison with *level-by-level projection*, bi-level projection reduces the cost of database projection and leads to improved performance when the database is huge and the support threshold is low.

A systematic performance study in [11] shows that PrefixSpan with these two optimizations is efficient and scalable. It mines the complete set of patterns and runs considerably faster than both Apriori-based GSP algorithm [12] and FreeSpan [6].

3 Performance Study

In this section, we present experimental results on the performance of GSP, FreeSpan, and PrefixSpan. It shows that PrefixSpan outperforms other previously proposed methods and is efficient and scalable for mining sequential patterns in large databases.

All the experiments are performed on a 233MHz Pentium PC machine with 128 megabytes main memory, running Microsoft Windows/NT. All the methods are implemented using Microsoft Visual C++ 6.0.

We compare performance of four methods as follows.

- GSP. The GSP algorithm was implemented as described in [12].
- FreeSpan. As reported in [6], FreeSpan with alternative level projection is more efficient than FreeSpan with level-by-level projection. In this paper, FreeSpan with alternative level projection is used.
- PrefixSpan-1. PrefixSpan-1 is PrefixSpan with level-by-level projection.
- PrefixSpan-2. PrefixSpan-2 is PrefixSpan with bi-level projection.

The synthetic datasets we used for our experiments were generated using standard procedure described in [2]. The same data generator has been used in most studies on sequential pattern mining, such as [12, 6]. We refer readers to [2] for more details on the generation of data sets.

We test the four methods on various datasets. The results are consistent. Limited by space, we report here only the results on dataset $C10T8S8I8$. In this data set, the number of items is set to $1,000$, and there are $10,000$ sequences in the data set. The average number of items within elements is set to 8 (denoted as $T8$). The average number of elements in a sequence is set to 8 (denoted as $S8$). There are a good number of long sequential patterns in it at low support thresholds.

The experimental results of scalability with support threshold are shown in Figure 2. When the support threshold is high, there are only a limited number of sequential patterns, and the length of patterns is short, the four methods are close in terms of runtime. However, as the support threshold decreases, the gaps become clear. Both FreeSpan and PrefixSpan win GSP. PrefixSpan methods are more efficient and more scalable than FreeSpan, too. Since the gaps among FreeSpan and GSP are clear, we focus on performance of various PrefixSpan techniques in the remaining of this section.

As shown in Figure 2, the performance curves of PrefixSpan-1 and PrefixSpan-2 are close when support threshold is not low. When the support threshold is low, since there are many sequential patterns, PrefixSpan-1 requires a major effort to generate projected databases. Bi-level projection can leverage the problem efficiently. As can be seen from Figure 3, the increase of runtime for PrefixSpan-2 is moderate even when the support threshold is pretty low.

Figure 3 also shows that using pseudo-projections for the projected databases that can be held in main memory improves efficiency of PrefixSpan further. As can be seen from the figure, the performance of level-by-level and bi-level pseudo-projections are close. Bi-level one catches up with level-by-level one when support threshold is very low. When the saving of less projected databases overcomes the cost of for mining and filling the *S-matrix*, bi-level projection wins. That verifies our analysis of level-by-level and bi-level projection.

Since pseudo-projection improves performance when

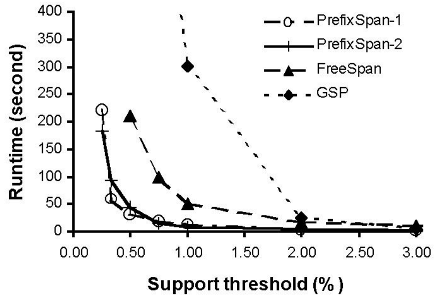

Figure 2: PrefixSpan, FreeSpan and GSP on data set $C10T8S8I8$.

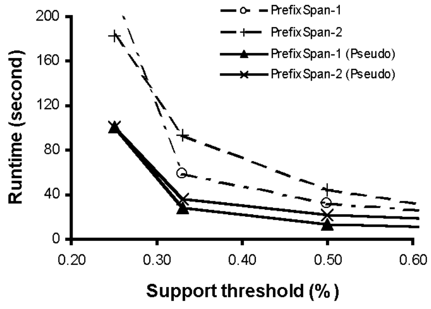

Figure 3: PrefixSpan and PrefixSpan (pseudo-proj) on data set $C10T8S8I8$.

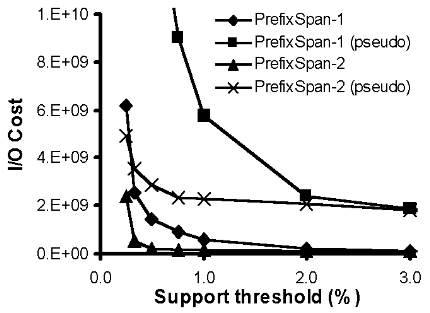

Figure 4: PrefixSpan and PrefixSpan (pseudo-proj) on large data set $C1kT8S8I8$.

the projected database can be held in main memory, a related question becomes: "can such a method be extended to disk-based processing?" That is, instead of doing physical projection and saving the projected databases in hard disk, should we make the projected database in the form of disk address and offset? To explore such an alternative, we pursue a simulation test as follows.

Let each sequential read, i.e., reading bytes in a data file from the beginning to the end, cost 1 unit of I/O. Let each random read, i.e., reading data according to its offset in the file, cost 1.5 unit of I/O. Also, suppose a write operation cost 1.5 I/O. Figure 4 shows the I/O costs of PrefixSpan-1 and PrefixSpan-2 as well as of their pseudo-projection variations over data set $C1kT8S8I8$ (where $C1k$ means 1 million sequences in the data set). PrefixSpan-1 and PrefixSpan-2 win their pseudo-projection variations clearly. It can also be observed that bi-level projection wins level-by-level projection as the support threshold becomes low. The huge number of random reads in disk-based pseudo-projections is the performance killer when the database is too big to fit into main memory.

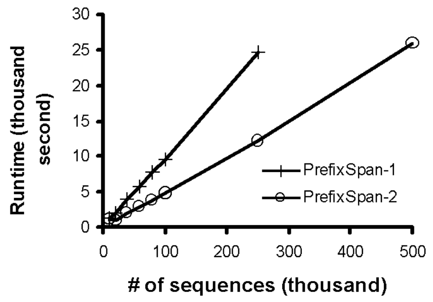

Figure 5: Scalability of PrefixSpan.

Figure 5 shows the scalability of PrefixSpan-1 and PrefixSpan-2 with respect to the number of sequences. Both methods are linearly scalable. Since the support threshold is set to 0.20%, PrefixSpan-2 performs better.

In summary, our performance study shows that PrefixSpan is more efficient and scalable than FreeSpan and GSP, whereas FreeSpan is faster than GSP when the support threshold is low, and there are many long patterns. Since PrefixSpan-2 uses bi-level projection to dramatically reduce the number of projections, it is more efficient than PrefixSpan-1 in large databases with low support threshold. Once the projected databases can be held in main memory, pseudo-projection always leads to the most efficient solution. The experimental results are consistent with our theoretical analysis.

4 Extending Pattern-growth Methods to Mine Sequential Patterns with Regular Expression Constraints

Sequential pattern mining often generates a large number of patterns, which reduces not only the efficiency but also the effectiveness of mining, since users have to sift through a large number of patterns to find useful ones.

Recent work has highlighted the importance of the paradigm of constraint-based mining: a user is allowed to express her focus in mining by means of a rich class of constraints capturing application semantics. Besides allowing user exploration and control, the paradigm allows many of these constraints to be pushed deep inside mining, thus pruning the search and achieving high performance.

Garofalakis et al. proposed the problem of sequential pattern mining with regular expression constraints in [4]. A regular expression constraint (*RE-constraint in short*) is specified as a regular expression over the set of items using regular expression operators, such as disjunction (|) and Kleene closure ($*$). Given a regular expression R, there exists a deterministic finite automation $A(R)$ such that $A(R)$ accepts exactly the language generated by R. A sequence s is said *satisfy* an RE-constraint R if every state transition in $A(R)$ is defined when following the sequence of transitions for the elements of s from the start state.

For example, $R = a * (bb|bcd|dd)$ is a regular expression over the set of items $\{a, b, c, d\}$. The deterministic finite automation for R is shown in Figure 6.

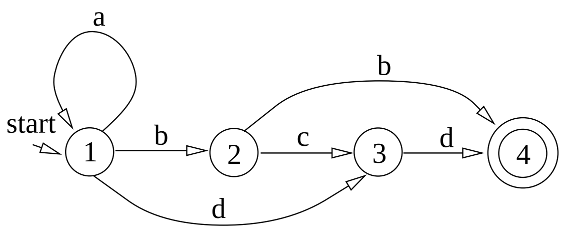

Figure 6: Deterministic finite automation for regular expression $R = a * (bb|bcd|dd)$.

As can be seen, on one hand, regular expressions have a simple as well as natural syntax for the succinct specification of sequential patterns. On the other hand, the expressive power of regular expressions is often enough for specifying a wide range of interesting and non-trivial patterns.

In [4], SPIRIT, a group of algorithms has been proposed. SPIRIT is based on extensions of Apriori-like methods. Since pattern-growth methods are more efficient and scalable than Apriori-like methods, a natural question is, "*can we extend pattern-growth methods to mine sequential patterns with RE-constraints*?

Let us take the fastest pattern-growth method, PrefixSpan, as the seed algorithm. We have the following property on satisfiability of patterns with respect to a deterministic finite automation.

Theorem 4.1 *Given an ER-constraint R. If following a sequence s of transitions from the start state in a deterministic finite automation $A(R)$ leads to a void state, any sequence s' with s as a prefix cannot satisfy constraint R.* $\square$

For example, sequence ac is not valid with respect to the deterministic automation $A(R)$ shown in Figure 6, neither is any sequence with ac as a prefix, such as acd. Therefore, none of them can satisfy the constraint.

Based on Theorem 4.1, PrefixSpan can be extended to find patterns satisfying RE-constraints. **Any patterns failing the RE-constraint should be pruned immediately. Only patterns valid so far are grown further.** The extended PrefixSpan prunes invalid patterns much early. For example, an Apriorix-like method dares not prune patterns ac or acd, even the pattern violates the automation. Otherwise, it may have trouble in assembling patterns $abcd$, which satisfies the RE-constraint.

The mining of sequential patterns with RE-constraint in PrefixSpan can be further optimized. For example, when mining with RE-constraint $a * (bb|bcd|dd)$, any other items except for a, b, c, and d should be removed from $\langle a \rangle$-projected database, since a, b, c, and d are the only items can be used to grow longer and legal patterns satisfying the constraint. By removing all other items, the database is shrunk and the search space becomes much smaller.

To explore which items should be included into the projected databases, we can trace back the automation from the accept state(s), e.g., state 4 in Figure 6. For the accept state, the *set of feasible items* is $\emptyset$. For each other state st, the set of feasible items of st is the union of sets of feasible items of states st adjacent to, as well as the items labeled on the outgoing edges. For example, in Figure 6, the sets of feasible items of state 4, 3, 2, and 1 are $\emptyset$, $\{d\}$, $\{b, c, d\}$, and $\{a, b, c, d\}$, respectively. Obviously, by browsing the automation only once, all sets of feasible items in every state can be determined.

Based on the above discussion, it is easy to give the details of extended PrefixSpan for mining sequential patterns with RE-constraints. Limited by space, we leave it as an exercise for interested readers.

5 Conclusions

We have presented a pattern-growth methodoloty for efficient sequential pattern mining in large sequence databases. Our performance study shows that pattern-growth methods are more efficient and scalable than Apriori-like methods. The high performance of pattern-growth methods is due to the following factors.

1. Pattern-growth methods adopt a divide-and-conquer strategy to project and partition a large database recursively into a set of progressively smaller ones, and the patterns to be searched for in each projected database are also reduced substantially.

2. Pattern-growth methods integrate disk-based data structures and fast in-memory traversal methods, which can be well-tuned to achieve high doing pseudo-projections.

3. Pattern-growth methods makes good use of the Apriori property implicitly, but avoids generating a large number of candidates, which ensures each counting and testing is on the real data sets rather than on the potential candidate sets.

There are many other kinds of interesting temporal patterns. As an example, we show that pattern-growth method PrefixSpan can be easily extended to mine sequential patterns with regular expression constraints. We believe that further extensions of pattern-growth methods are promising in mining other kinds of patterns, such as episodes [9], partial periodicity [5], etc. They are the interesting issues for future studies.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 1994 Int. Conf. Very Large Data Bases (VLDB'94)*, pages 487–499, Santiago, Chile, Sept. 1994.

[2] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. 1995 Int. Conf. Data Engineering (ICDE'95)*, pages 3–14, Taipei, Taiwan, Mar. 1995.

[3] C. Bettini, X. S. Wang, and S. Jajodia. Mining temporal relationships with multiple granularities in time sequences. *Data Engineering Bulletin*, 21:32–38, 1998.

[4] M. Garofalakis, R. Rastogi, and K. Shim. Spirit: Sequential pattern mining with regular expression constraints. In *Proc. 1999 Int. Conf. Very Large Data Bases (VLDB'99)*, pages 223–234, Edinburgh, UK, Sept. 1999.

[5] J. Han, G. Dong, and Y. Yin. Efficient mining of partial periodic patterns in time series database. In *Proc. 1999 Int. Conf. Data Engineering (ICDE'99)*, pages 106–115, Sydney, Australia, April 1999.

[6] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu. FreeSpan: Frequent pattern-projected sequential pattern mining. In *Proc. 2000 Int. Conf. Knowledge Discovery and Data Mining (KDD'00)*, pages 355–359, Boston, MA, Aug. 2000.

[7] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. 2000 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'00)*, pages 1–12, Dallas, TX, May 2000.

[8] H. Lu, J. Han, and L. Feng. Stock movement and n-dimensional inter-transaction association rules. In *Proc. 1998 SIGMOD Workshop Research Issues on Data Mining and Knowledge Discovery (DMKD'98)*, pages 12:1–12:7, Seattle, WA, June 1998.

[9] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1:259–289, 1997.

[10] B. Özden, S. Ramaswamy, and A. Silberschatz. Cyclic association rules. In *Proc. 1998 Int. Conf. Data Engineering (ICDE'98)*, pages 412–421, Orlando, FL, Feb. 1998.

[11] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proc. 2001 Int. Conf. Data Engineering (ICDE'01)*, pages 215–224, Heidelberg, Germany, April 2001.

[12] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. 5th Int. Conf. Extending Database Technology (EDBT'96)*, pages 3–17, Avignon, France, Mar. 1996.