1

Sequential Pattern Mining by Pattern-Growth: Principles and Extensions *

Jiawei Han[1], Jian Pei[2], and Xifeng Yan[1]

[1] University of Illinois at Urbana-Champaign {hanj, xyan}@cs.uiuc.edu
[2] State University of New York at Buffalo jianpei@cse.buffalo.edu

Summary. Sequential pattern mining is an important data mining problem with broad applications. However, it is also a challenging problem since the mining may have to generate or examine a combinatorially explosive number of intermediate subsequences. Recent studies have developed two major classes of sequential pattern mining methods: (1) a *candidate generation-and-test* approach, represented by (i) GSP [29], a horizontal format-based sequential pattern mining method, and (ii) SPADE [35], a vertical format-based method; and (2) a *sequential pattern growth* method, represented by PrefixSpan [25] and its further extensions, such as CloSpan for mining closed sequential patterns [34].

In this study, we perform a systematic introduction and presentation of the pattern-growth methodology and study its principles and extensions. We first introduce two interesting pattern growth algorithms, FreeSpan [11] and PrefixSpan [25], for efficient sequential pattern mining. Then we introduce CloSpan for mining closed sequential patterns. Their relative performance in large sequence databases is presented and analyzed. The various kinds of extension of these methods for (1) mining constraint-based sequential patterns, (2) mining multi-level, multi-dimensional sequential patterns, (3) mining top-k closed sequential patterns, and (4) their applications in bio-sequence pattern analysis and clustering sequences are also discussed in the paper.

Index terms. Data mining, sequential pattern mining algorithm, sequence database, scalability, performance analysis, application.

* The work was supported in part by the Natural Sciences and Engineering Research Council of Canada, the Networks of Centers of Excellence of Canada, the Hewlett-Packard Lab, the U.S. National Science Foundation NSF IIS-02-09199, NSF IIS-03-08001, and the University of Illinois. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

1.1 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 analysis of customer purchase patterns or Web access patterns, the analysis of sequencing or time-related processes such as scientific experiments, natural disasters, and disease treatments, the analysis of DNA sequences, and so on.

The sequential pattern mining problem was first introduced by Agrawal and Srikant in [2] based on their study of customer purchase sequences, as follows: *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 frequent subsequences, i.e., the subsequences whose occurrence frequency in the set of sequences is no less than min_support.*

This problem is introduced from the examination of potential patterns in sequence databases, as follows.

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. 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. For brevity, the brackets are omitted if an element has only one item, i.e., 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 $a_1 \subseteq b_{j_1}$, $a_2 \subseteq b_{j_2}$, $\ldots$, $a_n \subseteq b_{j_n}$.

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

Example 1. Let our running sequence database be S given in Table 1.1 and $min\_support = 2$. The set of *items* in the database is $\{a, b, c, d, e, f, g\}$.

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 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

Sequence_id	Sequence
1	$\langle a(abc)(ac)d(cf)\rangle$
2	$\langle (ad)c(bc)(ae)\rangle$
3	$\langle (ef)(ab)(df)cb\rangle$
4	$\langle eg(af)cbc\rangle$

Table 1.1. A sequence database

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*). □

From this example, one can see that sequential pattern mining problem can be stated as "*given a sequence database and the min_support threshold,* ***sequential pattern mining*** *is to find the complete set of sequential patterns in the database.*"

Notice that this model of sequential pattern mining is an abstraction from the customer shopping sequence analysis. However, this model may not cover a large set of requirements in sequential pattern mining. For example, for studying Web traversal sequences, gaps between traversals become important if one wants to predict what could be the next Web pages to be clicked. Many other applications may want to find gap-free or gap-sensitive sequential patterns as well, such as weather prediction, scientific, engineering and production processes, DNA sequence analysis, and so on. Moreover, one may like to find approximate sequential patterns instead of precise sequential patterns, such as in DNA sequence analysis where DNA sequences may contain nontrivial proportions of insertions, deletions, and mutations.

In our model of study, the gap between two consecutive elements in a sequence is unimportant. However, the gap-free or gap-sensitive frequent sequential patterns can be treated as special cases of our model since gaps are essentially constraints enforced on patterns. The efficient mining of gap-sensitive patterns will be discussed in our later section on constraint-based sequential pattern mining. Moreover, the mining of approximate sequential patterns is also treated as an extension of our basic mining methodology. Those and other related issues will be discussed in the later part of the paper.

Many previous studies contributed to the efficient mining of sequential patterns or other frequent patterns in time-related data [2, 29, 20, 31, 36, 21, 19, 4, 23, 28, 8]. Srikant and Agrawal [29] generalized their definition of sequential patterns in [2] to include time constraints, sliding time window, and user-defined taxonomy and present an Apriori-based, improved algorithm GSP (i.e., *generalized sequential patterns*). Mannila, et al. [20] presented a problem of mining frequent episodes in a sequence of events, where episodes are essentially acyclic graphs of events whose edges specify the tem-

poral precedent-subsequent relationship without restriction on interval. Bettini, et al. [4] considered a generalization of inter-transaction association rules. These are essentially rules whose left-hand and right-hand sides are episodes with time-interval restrictions. Lu, et al. [19] proposed inter-transaction association rules that are implication rules whose two sides are totally-ordered episodes with timing-interval restrictions. Garofalakis, et al. [7] proposed the use of regular expressions as a flexible constraint specification tool that enables user-controlled focus to be incorporated into the sequential pattern mining process. Some other studies extended the scope from mining sequential patterns to mining partial periodic patterns. Özden, et al. [23] introduced cyclic association rules that are essentially partial periodic patterns with *perfect* periodicity in the sense that *each pattern reoccurs in every cycle*, with 100% confidence. Han, et al. [8] developed a frequent pattern mining method for mining partial periodicity patterns that are frequent maximal patterns where each pattern appears in a fixed period with a fixed set of offsets, and with sufficient support. Zaki [35] developed a vertical format-based sequential pattern mining method, called SPADE, which can be considered as an extension of vertical-format-based frequent itemset mining methods, such as [36, 38].

Almost all of the above proposed methods for mining sequential patterns and other time-related frequent patterns are Apriori-like, i.e., based on the Apriori principle, which states the fact that *any super-pattern of an infrequent pattern cannot be frequent*, and based on a candidate generation-and-test paradigm proposed in association mining [1].

In our recent studies, we have developed and systematically explored a pattern-growth approach for efficient mining of sequential patterns in large sequence database. The approach adopts a divide-and-conquer, pattern-growth principle as follows, *sequence databases are recursively projected into a set of smaller projected databases based on the current sequential pattern(s), and sequential patterns are grown in each projected database by exploring only locally frequent fragments.* Based on this philosophy, we first proposed a straightforward pattern growth method, FreeSpan (for **Fre**qu**e**nt pattern-projected **S**equential **pa**ttern mining) [11], which reduces the efforts of candidate subsequence generation. Then, we introduced another and more efficient method, called PrefixSpan (for **Prefix**-projected **S**equential **pa**ttern mining), which offers ordered growth and reduced projected databases. To further improve the performance, a *pseudo-projection* technique is developed in PrefixSpan. A comprehensive performance study shows that PrefixSpan in most cases outperforms the Apriori-based GSP algorithm, FreeSpan, and SPADE [35] (a sequential pattern mining algorithm that adopts vertical data format), and PrefixSpan integrated with pseudo-projection is the fastest among all the tested algorithms. Furthermore, our experiments show that PrefixSpan consumes a much smaller memory space in comparison with GSP and SPADE.

PrefixSpan is an efficient algorithm at mining the complete set of sequential patterns. However, a long sequential pattern may contain a combinatorial number of frequent subsequences. To avoid generating a large number by

many of which are essentially redundant subsequences, our task becomes the mining of closed sequential pattern instead of the complete set of sequential patterns. An efficient algorithm called CloSpan [34] is developed based on the philosophy of (sequential) pattern-growth and by exploring sharing among generated or to be generated sequences. Our performance study shows that CloSpan may further reduce the cost at mining closed sequential patterns substantially in comparison with PrefixSpan.

This pattern-growth methodology has been further extended in various ways to cover the methods and applications of sequential and structured pattern mining. This includes (1) mining multi-level, multi-dimensional sequential patterns, (2) mining other structured patterns, such as graph patterns, (3) constraint-based sequential pattern mining, (4) mining closed sequential patterns, (5) mining top-k sequential patterns, (6) mining long sequences in the noise environment, (7) mining approximate consensus sequential patterns, and (8) clustering time-series gene expressions.

In this paper, we will systematically present the methods for pattern-growth-based sequential patterns, their principle and applications.

The remainder of the paper is organized as follows. In Section 2, we introduce the Apriori-based sequential pattern mining methods, GSP and SPADE, both relying on a candidate generation-and-test philosophy. In Section 3, our approach, projection-based sequential pattern growth, is introduced, by first summarizing FreeSpan, and then presenting PrefixSpan, associated with a pseudo-projection technique for performance improvement. In Section 4, we introduce CloSpan, an efficient method for mining closed sequential patterns. Some experimental results and performance analysis are summarized in Section 5. The extensions of the method in different directions are discussed in Section 6. We conclude our study in Section 7.

1.2 Previous work: The Candidate Generation-and-Test Approach

The candidate generation-and-test approach is an extension of the Apriori-based frequent pattern mining algorithm [1] to sequential pattern analysis. Similar to frequent patterns, sequential patterns has the anti-monotone (i.e., downward closure) property as follows: *every non-empty sub-sequence of a sequential pattern is a sequential pattern.*

Based on this property, there are two algorithms developed for efficient sequential pattern mining: (1) a horizontal data format based sequential pattern mining method: GSP [29], and (2) a vertical data format based sequential pattern mining method: SPADE [35]. We outline and analyze these two methods in this section.

1.2.1 GSP: A horizontal data format based sequential pattern mining algorithm

From the sequential pattern mining point of view, a sequence database can be represented in two data formats: (1) a horizontal data format, and (2) a vertical data format. The former uses the natural representation of the data set as $\langle sequence\_id : a\_sequence\_of\_objects \rangle$, whereas the latter uses the vertical representation of the sequence database: $\langle object : (sequence\_id, time\_stamp) \rangle$, which can be obtained by transforming from a horizontal formatted sequence database.

GSP is a horizontal data format based sequential pattern mining developed by Srikant and Agrawal [29] by extension of their frequent itemset mining algorithm, Apriori [1]. Based on the downward closure property of a sequential pattern, GSP adopts a multiple-pass, candidate-generation-and-test approach in sequential pattern mining. The algorithm 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.

The method is illustrated using 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, i.e., frequent length-1 subsequences (in the form of "*item* : *support*"): $\langle a \rangle : 4, \langle b \rangle : 4, \langle c \rangle : 3, \langle d \rangle : 3, \langle e \rangle : 3, \langle f \rangle : 3, \langle g \rangle : 1$.

By filtering the infrequent item 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 member in the set 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 $6 \times 6 + \frac{6\times5}{2} = 51$ 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 1.1. 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-$(k-1)$

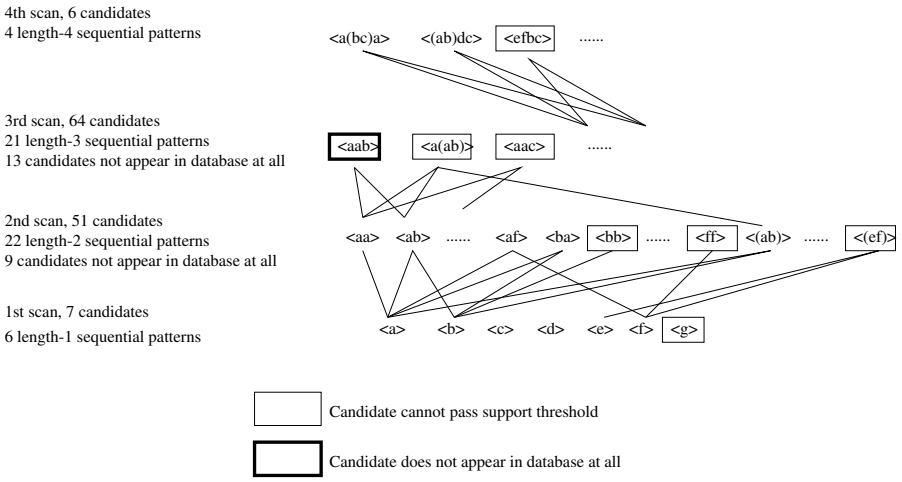


Fig. 1.1. Candidates, candidate generation, and sequential patterns in GSP

subsequences is a sequential pattern found at the $(k-1)$-th 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. Clearly, 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. Some 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. □

The example shows that an Apriori-like sequential pattern mining method, such as GSP, though reduces search space, bears three nontrivial, inherent costs which are independent of detailed implementation techniques.

First, *there are potentially huge sets 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 $\langle a_1\rangle$, $\langle a_2\rangle$, $\ldots$, $\langle a_{1000}\rangle$, 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 $\langle a_1a_1\rangle$, $\langle a_1a_2\rangle$, $\ldots$, $\langle a_1a_{1000}\rangle$, $\langle a_2a_1\rangle$, $\langle a_2a_2\rangle$, $\ldots$, $\langle a_{1000}a_{1000}\rangle$, and the second term is derived from the set $\langle(a_1a_2)\rangle$, $\langle(a_1a_3)\rangle$, $\ldots$, $\langle(a_{999}a_{1000})\rangle$.

Second, *multiple scans of databases could be costly.* 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.

Last, *there are inherent 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, $\binom{100}{3} = 161,700$ length-3 candidate sequences, and so on. Obviously, the total number of candidate sequences to be generated is greater than $\Sigma_{i=1}^{100} \binom{100}{i} = 2^{100} - 1 \approx 10^{30}$.

In many applications, it is not rare 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. Then the problem becomes, "*can we develop a method which may absorb the spirit of* Apriori *but avoid or substantially reduce the expensive candidate generation and test?*"

1.2.2 SPADE: An Apriori-based vertical data format sequential pattern mining algorithm

The Apriori-based sequential pattern mining can also be explored by mapping a sequence database into the vertical data format which takes each item as the center of observation and takes its associated sequence and event identifiers as data sets. To find sequence of length-2 items, one just needs to join two single items if they are frequent and they share the same sequence identifier and their event identifiers (which are essentially relative timestamps) follow the sequential ordering. Similarly, one can grow the length of itemsets from length two to length three, and so on. Such an Apriori-based vertical data format sequential pattern mining algorithm, called SPADE (**S**equential **PA**ttern **D**iscovery using **E**quivalent classes) algorithm [35], is illustrated using the following example.

Example 3. (SPADE) Given our running sequence database S and *min_support* in Example 1, SPADE first scans S, transforms the database into the vertical format by introducing EID (event ID) which is a (local) timestamp for

each event. Each single item is associated with a set of SID (sequence_id) and EID (event_id) pairs. For example, item "b" is associated with (SID, EID) pairs as follows: $\{(1,2),(2,3),(3,2),(3,5),(4,5)\}$, as shown in Figure 1.2. This is because item b appears in sequence 1, event 2, and so on. Frequent single items "a" and "b" can be joined together to form a length-two subsequence by joining the same sequence_id with event_ids following the corresponding sequence order. For example, subsequence ab contains a set of triples $(SID, EID(a), EID(b))$, such as $(1,1,2)$, and so on. Furthermore, the frequent length-2 subsequences can be joined together based on the Apriori heuristic to form length-3 subsequences, and so on. The process continuous until no frequent sequences can be found or no such sequences can be formed by such joins.

SID	EID	Items
1	1	a
1	2	abc
1	3	ac
1	4	d
1	5	cf
2	1	ad
2	2	c
2	3	bc
2	4	ae
3	1	ef
3	2	ab
3	3	df
3	4	c
3	5	b
4	1	e
4	2	g
4	3	af
4	4	c
4	5	b
4	6	c

a		b		···
SID	EID	SID	EID	···
1	1	1	2	
1	2	2	3	
1	3	3	2	
2	1	3	5	
2	4	4	5	
3	2			
4	3			

ab			ba			···
SID	EID (a)	EID(b)	SID	EID (b)	EID(a)	···
1	1	2	1	2	3	
2	1	3	2	3	4	
3	2	5				
4	3	5				

aba				···
SID	EID (a)	EID(b)	EID(a)	···
1	1	2	3	
2	1	3	4	

Fig. 1.2. Vertical format of the sequence database and fragments of the SPADE mining process.

Some fragments of the SPADE mining process are illustrated in Figure 1.2. The detailed analysis of the method can be found in [35]. □

The SPADE algorithm may reduce the access of sequence databases since the information required to construct longer sequences are localized to the related items and/or subsequences represented by their associated sequence

and event identifiers. However, the basic search methodology of SPADE is similar to GSP, exploring both breadth-first search and Apriori pruning. It has to generate a large set of candidates in breadth-first manner in order to grow longer subsequences. Thus most of the difficulties suffered in the GSP algorithm will reoccur in SPADE as well.

1.3 The Pattern-growth Approach for Sequential Pattern Mining

In this section, we introduce a pattern-growth methodology for mining sequential patterns. It is based on the methodology of pattern-growth mining of frequent patterns in transaction databases developed in the FP-growth algorithm [12]. We introduce first the FreeSpan algorithm and then a more efficient alternative, the PrefixSpan algorithm.

1.3.1 FreeSpan: Frequent Pattern-Projected Sequential Pattern Mining

For a sequence $\alpha = \langle s_1 \cdots s_l \rangle$, the itemset $s_1 \cup \cdots \cup s_l$ is called α's *projected itemset*. FreeSpan is based on the following property: *if an itemset X is infrequent, any sequence whose projected itemset is a superset of X cannot be a sequential pattern*. FreeSpan mines sequential patterns by partitioning the search space and projecting the sequence sub-databases recursively based on the projected itemsets.

Let f_list $= \langle x_1, \ldots, x_n \rangle$ be a list of all frequent items in sequence database S. Then, the complete set of sequential patterns in S can be divided into n disjoint subsets: (1) the set of sequential patterns containing only item x_1, (2) those containing item x_2 but no item in $\{x_3, \ldots, x_n\}$, and so on. In general, the i^{th} subset $(1 \leq i \leq n)$ is the set of sequential patterns containing item x_i but no item in $\{x_{i+1}, \ldots, x_n\}$.

Then, the database projection can be performed as follows. At the time of deriving p's projected database from DB, the set of frequent items X of DB is already known. Only those items in X will need to be projected into p's projected database. This effectively discards irrelevant information and keeps the size of the projected database minimal. By recursively doing so, one can mine the projected databases and generate the complete set of sequential patterns in the given partition without duplication. The details are illustrated in the following example.

*Example 4 (*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. This step is similar to GSP. Frequent items are listed in support descending order (in the form of "*item* : *support*"), that is, f_list =

$\langle a:4,\ b:4,\ c:4,\ d:3,\ e:3,\ f:3\rangle$. They form six length-one sequential patterns: $\langle a\rangle$:4, $\langle b\rangle$:4, $\langle c\rangle$:4, $\langle d\rangle$:3, $\langle e\rangle$:3, $\langle f\rangle$:3.

According to the 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 no item after b in f_list, (3) the ones containing item c but no item after c in f_list, and so on, and finally, (6) the ones containing item f.

The sequential patterns related to the six partitioned subsets can be mined by constructing six *projected databases* (obtained by one additional scan of the original database). Infrequent items, such as g in this example, are removed from the projected databases. The process for mining each projected database is detailed as follows.

- *Mining sequential patterns containing only item a.*
  The $\langle a\rangle$*-projected database* is $\{\langle aaa\rangle, \langle aa\rangle, \langle a\rangle, \langle a\rangle\}$. By mining this projected database, only one additional sequential pattern containing only item a, i.e., $\langle aa\rangle$:2, is found.
- *Mining sequential patterns containing item b but no item after b in the* f_list.
  By mining the $\langle b\rangle$*-projected database*: $\{\langle a(ab)a\rangle, \langle aba\rangle, \langle (ab)b\rangle, \langle ab\rangle\}$, four additional sequential patterns containing item b but no item after b in f_list are found. They are $\{\langle ab\rangle$:4, $\langle ba\rangle$:2, $\langle (ab)\rangle$:2, $\langle aba\rangle$:2$\}$.
- *Mining sequential patterns containing item c but no item after c in the* f_list.
  The mining of the $\langle c\rangle$*-projected database*: $\{\langle a(abc)(ac)c\rangle, \langle ac(bc)a\rangle, \langle (ab)cb\rangle, \langle acbc\rangle\}$, proceeds as follows. One scan of the projected database generates the set of length-2 frequent sequences, which are $\{\langle ac\rangle$:4, $\langle (bc)\rangle$:2, $\langle bc\rangle$:3,$\langle cc\rangle$:3, $\langle ca\rangle$:2, $\langle cb\rangle$:3$\}$. One additional scan of the $\langle c\rangle$*-projected database* generates all of its projected databases.
  The mining of the $\langle ac\rangle$*-projected database*: $\{\langle a(abc)(ac)c\rangle, \langle ac(bc)a\rangle, \langle (ab)cb\rangle, \langle acbc\rangle\}$ generates the set of length-3 patterns as follows: $\{\langle acb\rangle$:3, $\langle acc\rangle$:3, $\langle (ab)c\rangle$:2, $\langle aca\rangle$:2$\}$. Four projected database will be generated from them. The mining of the first one, the $\langle acb\rangle$*-projected database*: $\{\langle ac(bc)a\rangle, \langle (ab)cb\rangle, \langle acbc\rangle\}$ generates no length-4 pattern. The mining along this line terminates. Similarly, we can show that the mining of the other three projected databases terminates without generating any length-4 patterns for the $\langle ac\rangle$*-projected database.*
- *Mining other subsets of sequential patterns.*
  Other subsets of sequential patterns can be mined similarly on their corresponding projected databases. This mining process proceeds recursively, which derives the complete set of sequential patterns. □

The detailed presentation of the FreeSpan algorithm, the proof of its completeness and correctness, and the performance study of the algorithm are in [11]. By the analysis of Example 4 and verified by our experimental study, we have the following observations on the strength and weakness of FreeSpan:

The strength of FreeSpan *is that it searches a smaller projected database than* GSP *in each subsequent database projection.* This is because that FreeSpan projects a large sequence database recursively into a set of small projected sequence databases based on the currently mined frequent item-patterns, and the subsequent mining is confined to each projected database relevant to a smaller set of candidates.

The major overhead of FreeSpan *is that it may have to generate many non-trivial 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 contains three same sequences as that in the original sequence database, except for the removal of the infrequent item g in sequence 4. 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.

1.3.2 PrefixSpan: Prefix-Projected Sequential Patterns Mining

Based on the analysis of the FreeSpan algorithm, one can see that one may still have to pay high cost at handling projected databases. To avoid checking every possible combination of a potential candidate sequence, one can first fix the order of items *within each element.* Since items within an element of a sequence can be listed in any order, without loss of generality, one can assume that they are always listed alphabetically. For example, the sequence in S with Sequence_id 1 in our running example is listed as $\langle a(abc)(ac)d(cf)\rangle$ instead of $\langle a(bac)(ca)d(fc)\rangle$. With such a convention, the expression of a sequence is unique.

Then, we examine whether one can fix the order of item projection in the generation of a projected database. Intuitively, if one follows the order of the prefix of a sequence and projects only the suffix of a sequence, one can examine in an orderly manner all the possible subsequences and their associated projected database. Thus we first introduce the concept of prefix and suffix.

Suppose all the items within an element are listed alphabetically. Given a sequence $\alpha = \langle e_1 e_2 \cdots e_n\rangle$ (where each e_i corresponds to a frequent element in S), a sequence $\beta = \langle e'_1 e'_2 \cdots e'_m\rangle$ $(m \leq n)$ is called a **prefix** of α if and only if (1) $e'_i = e_i$ for $(i \leq m-1)$; (2) $e'_m \subseteq e_m$; and (3) all the frequent items in $(e_m - e'_m)$ are alphabetically after those in e'_m. Sequence $\gamma = \langle e''_m e_{m+1} \cdots e_n\rangle$ is called the **suffix** of α w.r.t. prefix β, denoted as $\gamma = \alpha/\beta$, where $e''_m = (e_m - e'_m)$.[3] We also denote $\alpha = \beta \cdot \gamma$. Note if β is not a subsequence of α, the suffix of α w.r.t. β is empty.

Example 5. For a sequence $s = \langle a(abc)(ac)d(cf)\rangle$, $\langle a\rangle$, $\langle aa\rangle$, $\langle a(ab)\rangle$ and $\langle a(abc)\rangle$ are *prefixes* of sequence $s = \langle a(abc)(ac)d(cf)\rangle$, but neither $\langle ab\rangle$ nor

[3] If e''_m is not empty, the suffix is also denoted as $\langle(\_ \text{ items in } e''_m)e_{m+1}\cdots e_n\rangle$.

$\langle a(bc)\rangle$ is considered as a prefix if every item in the prefix $\langle a(abc)\rangle$ of sequence s is frequent in S. Also, $\langle(abc)(ac)d(cf)\rangle$ is the *suffix* w.r.t. the prefix $\langle a\rangle$, $\langle(\_bc)(ac)d(cf)\rangle$ is the *suffix* w.r.t. the prefix $\langle aa\rangle$, and $\langle(\_c)(ac)d(cf)\rangle$ is the *suffix* w.r.t. the prefix $\langle a(ab)\rangle$. □

Based on the concepts of prefix and suffix, the problem of mining sequential patterns can be decomposed into a set of subproblems as shown below.

1. Let $\{\langle x_1\rangle, \langle x_2\rangle, \ldots, \langle x_n\rangle\}$ be the complete set of length-1 sequential patterns in a sequence database S. The complete set of sequential patterns in S can be divided into n disjoint subsets. The i^{th} subset $(1 \leq i \leq n)$ is the set of sequential patterns with prefix $\langle x_i\rangle$.
2. Let α be a length-l sequential pattern and $\{\beta_1, \beta_2, \ldots, \beta_m\}$ be the set of all length-$(l+1)$ sequential patterns with prefix α. The complete set of sequential patterns with prefix α, except for α itself, can be divided into m disjoint subsets. The j^{th} subset $(1 \leq j \leq m)$ is the set of sequential patterns prefixed with β_j.

Based on this observation, the problem can be partitioned recursively. That is, each subset of sequential patterns can be further divided when necessary. This forms a *divide-and-conquer* framework. To mine the subsets of sequential patterns, the corresponding projected databases can be constructed.

Let α be a sequential pattern in a sequence database S. The **α-projected database**, denoted as $S|_\alpha$, is the collection of suffixes of sequences in S w.r.t. prefix α. Let β be a sequence with prefix α. The **support count** of β in α-projected database $S|_\alpha$, denoted as $support_{S|_\alpha}(\beta)$, is the number of sequences γ in $S|_\alpha$ such that $\beta \sqsubseteq \alpha \cdot \gamma$.

We have the following lemma regarding to the projected databases.

Lemma 1. (Projected database) *Let α and β be two sequential patterns in a sequence database S such that α is a prefix of β.*

1. *$S|_\beta = (S|_\alpha)|_\beta$;*
2. *for any sequence γ with prefix α, $support_S(\gamma) = support_{S|_\alpha}(\gamma)$; and*
3. *The size of α-projected database cannot exceed that of S.*

Proof sketch. The first part of the lemma follows the fact that, for a sequence γ, the suffix of γ w.r.t. β, γ/β, equals to the sequence resulted from first doing projection of γ w.r.t. α, i.e., γ/α, and then doing projection γ/α w.r.t. β. That is $\gamma/\beta = (\gamma/\alpha)/\beta$.

The second part of the lemma states that to collect support count of a sequence γ, only the sequences in the database sharing the same prefix should be considered. Furthermore, only those suffixes with the prefix being a super-sequence of γ should be counted. The claim follows the related definitions.

The third part of the lemma is on the size of a projected database. Obviously, the α-projected database can have the same number of sequences as S only if α appears in every sequence in S. Otherwise, only those sequences

in S which are super-sequences of α appear in the α-projected database. So, the α-projected database cannot contain more sequences than S. For every sequence γ in S such that γ is a super-sequence of α, γ appears in the α-projected database in whole only if α is a prefix of γ. Otherwise, only a subsequence of γ appears in the α-projected database. Therefore, the size of α-projected database cannot exceed that of S. □

Let us examine how to use the prefix-based projection approach for mining sequential patterns based our running example.

Example 6. (PrefixSpan) For the same sequence database S in Table 1.1 with $min\_sup = 2$, sequential patterns in S can be mined by a prefix-projection method in the following steps.

1. *Find length-1 sequential patterns.*
   Scan S once to find all the frequent items in sequences. Each of these frequent items is a length-1 sequential pattern. They are $\langle a\rangle : 4$, $\langle b\rangle : 4$, $\langle c\rangle : 4$, $\langle d\rangle : 3$, $\langle e\rangle : 3$, and $\langle f\rangle : 3$, where the notation "$\langle pattern\rangle : count$" represents the pattern and its associated support count.
2. *Divide search space.*
   The complete set of sequential patterns can be partitioned into the following six subsets according to the six prefixes: (1) the ones with prefix $\langle a\rangle$, (2) the ones with prefix $\langle b\rangle$, ..., and (6) the ones with prefix $\langle f\rangle$.

prefix	projected 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 1.2. Projected databases and sequential patterns

3. *Find subsets of sequential patterns.*
   The subsets of sequential patterns can be mined by constructing the corresponding set of *projected databases* and mining each recursively. The projected databases as well as sequential patterns found in them are listed in Table 1.2, while the mining process is explained as follows.

a) *Find sequential patterns with prefix* $\langle a \rangle$.
Only the sequences containing $\langle a \rangle$ should be collected. Moreover, in a sequence containing $\langle a \rangle$, only the subsequence prefixed with the first occurrence of $\langle a \rangle$ should be considered. For example, in sequence $\langle (ef)(ab)(df)cb \rangle$, only the subsequence $\langle (\_b)(df)cb \rangle$ should be considered for mining sequential patterns prefixed with $\langle a \rangle$. Notice that $(\_b)$ means that the last element in the prefix, which is a, together with b, form one element.
The sequences in S containing $\langle a \rangle$ are projected w.r.t. $\langle a \rangle$ to form the $\langle a \rangle$-*projected database*, which consists of four suffix 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 the $\langle a \rangle$-projected database once, its locally frequent items are $a : 2$, $b : 4$, $\_b : 2$, $c : 4$, $d : 2$, and $f : 2$. Thus all the length-2 sequential patterns prefixed with $\langle a \rangle$ are found, and 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) those prefixed with $\langle aa \rangle$, (2) those with $\langle ab \rangle$, ..., and finally, (6) those with $\langle af \rangle$. These subsets can be mined by constructing respective projected databases and mining each recursively as follows.

i. The $\langle aa \rangle$-projected database consists of two non-empty (suffix) subsequences prefixed with $\langle aa \rangle$: $\{\langle (\_bc)(ac)d(cf) \rangle, \{\langle (\_e) \rangle\}$. Since there is no hope to generate any frequent subsequence from this projected database, the processing of the $\langle aa \rangle$-projected database terminates.

ii. The $\langle ab \rangle$-projected database consists of three suffix sequences: $\langle (\_c)(ac)d(cf) \rangle$, $\langle (\_c)a \rangle$, and $\langle c \rangle$. Recursively mining the $\langle ab \rangle$-projected database 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 $\langle abc \rangle$.) They form the complete set of sequential patterns prefixed with $\langle ab \rangle$.

iii. The $\langle (ab) \rangle$-projected database contains only two sequences: $\langle (\_c)(ac)d(cf) \rangle$ and $\langle (df)cb \rangle$, which leads to the finding of the following sequential patterns prefixed with $\langle (ab) \rangle$: $\langle c \rangle$, $\langle d \rangle$, $\langle f \rangle$, and $\langle dc \rangle$.

iv. The $\langle ac \rangle$-, $\langle ad \rangle$- and $\langle af \rangle$- projected databases can be constructed and recursively mined similarly. The sequential patterns found are shown in Table 1.2.

b) *Find sequential patterns with prefix* $\langle b \rangle$, $\langle c \rangle$, $\langle d \rangle$, $\langle e \rangle$ *and* $\langle f \rangle$, *respectively.*
This can be done by constructing the $\langle b \rangle$-, $\langle c \rangle$- $\langle d \rangle$-, $\langle e \rangle$- and $\langle f \rangle$-projected databases and mining them respectively. The projected databases as well as the sequential patterns found are shown in Table 1.2.

4. *The set of sequential patterns is the collection of patterns found in the above recursive mining process.*

One can verify that it returns exactly the same set of sequential patterns as what GSP and FreeSpan do. □

Based on the above discussion, the algorithm of PrefixSpan is presented as follows.

Algorithm 1 (PrefixSpan) Prefix-projected sequential pattern mining.

Input: A sequence database S, and the minimum support threshold $min\_support$.
Output: The complete set of sequential patterns.
Method: Call PrefixSpan$(\langle\rangle, 0, S)$.

Subroutine PrefixSpan$(\alpha, l, S|_\alpha)$
The parameters are (1) α is a sequential pattern; (2) l is the length of α; and (3) $S|_\alpha$ is the α-projected database if $\alpha \neq \langle\rangle$, otherwise, it is the sequence database S.
Method:

1. Scan $S|_\alpha$ once, find each frequent item, b, such that
   a) b can be assembled to the last element of α to form a sequential pattern; or
   b) $\langle b\rangle$ can be appended to α to form a sequential pattern.
2. For each frequent item b, append it to α to form a sequential pattern α', and output α';
3. For each α', if α' is frequent, construct α'-projected database $S|_{\alpha'}$, and call PrefixSpan$(\alpha', l+1, S|_{\alpha'})$.

Analysis. The correctness and completeness of the algorithm can be justified based on Lemma 1. Here, we analyze the efficiency of the algorithm as follows.

- *No candidate sequence needs to be generated by* PrefixSpan.
  Unlike Apriori-like algorithms, PrefixSpan only grows longer sequential patterns from the shorter frequent ones. It neither generates nor tests any candidate sequence non-existent in a projected database. Comparing with GSP, which generates and tests a substantial number of candidate sequences, PrefixSpan searches a much smaller space.
- *Projected databases keep shrinking.*
  As indicated in Lemma 1, a projected database is smaller than the original one because only the suffix subsequences of a frequent prefix are projected into a projected database. In practice, the shrinking factors can be significant because (1) usually, only a small set of sequential patterns grow quite long in a sequence database, and thus the number of sequences in a projected database usually reduces substantially when prefix grows; and (2) projection only takes the suffix portion with respect to a prefix. Notice that FreeSpan also employs the idea of projected databases. However, the projection there often takes the whole string (not just suffix) and thus the shrinking factor is less than that of PrefixSpan.

- *The major cost of* PrefixSpan *is the construction of projected databases.* In the worst case, PrefixSpan constructs a projected database for every sequential pattern. If there exist a good number of sequential patterns, the cost is non-trivial. Techniques for reducing the number of projected databases will be discussed in the next subsection. □

1.3.3 Pseudo-Projection

The above analysis shows that the major cost of PrefixSpan is database projection, i.e., forming projected databases recursively. Usually, a large number of projected databases will be generated in sequential pattern mining. If the number and/or the size of projected databases can be reduced, the performance of sequential pattern mining can be further improved.

One technique which may reduce the number and size of projected databases is *pseudo-projection*. The idea is outlined as follows. Instead of performing physical projection, one can register the index (or identifier) of the corresponding sequence and the starting position of the projected suffix in the sequence. Then, a physical projection of a sequence is replaced by registering a sequence identifier and the projected position index point. *Pseudo-projection* reduces the cost of projection substantially when the projected database can fit in main memory.

This method is based on the following observation. For any sequence s, each projection can be represented by a corresponding projection position (an index point) instead of copying the whole suffix as a projected subsequence. Consider a sequence $\langle a(abc)(ac)d(cf)\rangle$. Physical projections may lead to repeated copying of different suffixes of the sequence. An index position pointer may save physical projection of the suffix and thus save both space and time of generating numerous physical projected databases.

Example 7. (Pseudo-projection) For the same sequence database S in Table 1.1 with $min\_sup = 2$, sequential patterns in S can be mined by pseudo-projection method as follows.

Suppose the sequence database S in Table 1.1 can be held in main memory. Instead of constructing the $\langle a\rangle$-projected database, one can represent the projected suffix sequences using pointer (sequence_id) and offset(s). For example, the projection of sequence $s_1 = \langle a(abc)d(ae)(cf)\rangle$ with regard to the $\langle a\rangle$-projection consists two pieces of information: (1) a *pointer* to s_1 which could be the string_id s_1, and (2) the *offset(s)*, which should be a single integer, such as 2, if there is a single projection point; and a set of integers, such as $\{2, 3, 6\}$, if there are multiple projection points. Each offset indicates at which position the projection starts in the sequence.

The projected databases for prefixes $\langle a\rangle$-, $\langle b\rangle$-, $\langle c\rangle$-, $\langle d\rangle$-, $\langle f\rangle$-, and $\langle aa\rangle$-are shown in Table 1.3, where \$ indicates the prefix has an occurrence in the current sequence but its projected suffix is empty, whereas $\emptyset$ indicates that there is no occurrence of the prefix in the corresponding sequence. From Table

Sequence_id	Sequence	$\langle a \rangle$	$\langle b \rangle$	$\langle c \rangle$	$\langle d \rangle$	$\langle f \rangle$	$\langle aa \rangle$	...
10	$\langle a(abc)(ac)d(cf) \rangle$	2, 3, 6	4	5, 7	8	\$	3, 6	...
20	$\langle (ad)c(bc)(ae) \rangle$	2	5	4, 6	3	$\emptyset$	7	...
30	$\langle (ef)(ab)(df)cb \rangle$	4	5	8	6	3, 7	$\emptyset$	...
40	$\langle eg(af)cbc \rangle$	4	6	6	$\emptyset$	5	$\emptyset$	...

Table 1.3. A sequence database and some of its pseudo-projected databases

1.3, one can see that the pseudo-projected database usually takes much less space than its corresponding physically projected one. □

Pseudo-projection avoids physically copying suffixes. Thus, it is efficient in terms of both running time and space. However, it may not be efficient if the pseudo-projection is used for disk-based accessing since random access disk space is costly. Based on this observation, the suggested approach is that if the original sequence database or the projected databases is too big to fit in memory, the physical projection should be applied, however, the execution should be swapped to pseudo-projection once the projected databases can fit in memory. This methodology is adopted in our PrefixSpan implementation.

Notice that the pseudo-projection works efficiently for PrefixSpan but not so for FreeSpan. This is because for PrefixSpan, an offset position clearly identifies the suffix and thus the projected subsequence. However, for FreeSpan, since the next step pattern-growth can be in both forward and backward directions, one needs to register more information on the possible extension positions in order to identify the remainder of the projected subsequences. Therefore, we only explore the pseudo-projection technique for PrefixSpan.

1.4 CloSpan: Mining Closed Frequent Sequential Patterns

The sequential pattern mining algorithms developed so far have good performance in databases consisting of short frequent sequences. Unfortunately, when mining long frequent sequences, or when using very low support thresholds, the performance of such algorithms often degrades dramatically. This is not surprising: Assume the database contains only one long frequent sequence $\langle (a_1)(a_2)\dots(a_{100}) \rangle$, it will generate $2^{100} - 1$ frequent subsequences if the minimum support is 1, although all of them except the longest one are redundant because they have the same support as that of $\langle (a_1)(a_2)\dots(a_{100}) \rangle$.

We propose an alternative but equally powerful solution: instead of mining the complete set of frequent subsequences, we mine frequent *closed subsequences* only, i.e., those containing no super-sequence with the same support. We develop CloSpan [34] (Closed Sequential pattern mining) to mine these patterns. CloSpan can produce a significantly less number of sequences than the traditional (i.e., full-set) methods while preserving the same expressive power since the whole set of frequent subsequences, together with their supports, can be derived easily from our mining results.

CloSpan first mines a closed sequence candidate set which contains all frequent closed sequences. The candidate set may contain some non-closed sequences. Thus, CloSpan needs a post-pruning step to filter out non-closed sequences. In order to efficiently mine the candidate set, we introduce a search space pruning condition: Whenever we find two exactly same prefix-based project databases, we can stop growing one prefix.

Let $\mathcal{I}(S)$ represent the total number of items in S, defined by

$$\mathcal{I}(S) = \sum_{\alpha \in S} l(\alpha),$$

where $l(\alpha)$ is α's length. We call $\mathcal{I}(S)$ the *the database size*. For the sample dataset in Table 1.1, $\mathcal{I}(S) = 31$.

Theorem 1 (Equivalence of Projected Databases). *Given two sequences, $\alpha \sqsubseteq \beta$, then*

$$S|_\alpha = S|_\beta \Leftrightarrow \mathcal{I}(S|_\alpha) = \mathcal{I}(S|_\beta) \tag{1.1}$$

Proof. *It is obvious that $S|_\alpha = S|_\beta \Rightarrow \mathcal{I}(S|_\alpha) = \mathcal{I}(S|_\beta)$. Now we prove the sufficient condition. Since $\alpha \sqsubseteq \beta$, then $\mathcal{I}(S|_\alpha) \leqslant \mathcal{I}(S|_\beta)$. The equality between $\mathcal{I}(S|_\alpha)$ and $\mathcal{I}(S|_\beta)$ holds only if $\forall \gamma \in S|_\beta$, $\gamma \in S|_\alpha$, and vice vera. Therefore, $S|_\alpha = S|_\beta$.* □

For the sample database in Table 1.1, $S|_{\langle ac \rangle} = S|_{\langle c \rangle} = \{\langle (ac)d(cf) \rangle, \langle (bc)(ae) \rangle, \langle b \rangle, \langle bc \rangle\}$ and $\mathcal{I}(S|_{\langle ac \rangle}) = \mathcal{I}(S|_{\langle c \rangle}) = 12$. According to Theorem 1, the search space can be pruned as follows.

Lemma 2 (Early Termination by Equivalence). *Given two sequences, $\alpha \sqsubseteq \beta$, if $\mathcal{I}(S|_\alpha) = \mathcal{I}(S|_\beta)$, then $\forall \gamma, support(\alpha \diamond \gamma) = support(\beta \diamond \gamma)$, where $\alpha \diamond \gamma$ and $\beta \diamond \gamma$ means γ is assembled to the last itemset of α and β (or appended to them), respectively.*

Considering the previous example, we have $\mathcal{I}(S|_{\langle ac \rangle}) = \mathcal{I}(S|_{\langle c \rangle})$. Based on Lemma 2, without calculating the supports of $\langle acb \rangle$ and $\langle cb \rangle$, we can conclude that they are the same.

The search space of PrefixSpan, when mining frequent sequences in Table 1.1, is depicted in Figure 1.3. Each node in the figure represents one frequent sequence. PrefixSpan performs depth-first search by assembling one item to the last itemset of the current frequent sequence or appending one item to it. In Figure 1.3, we use subscript "i" to denote the assembling extension, and "s" to denote the appending extension.

According to Lemma 2, it is recognized that if α and all of its descendants $(\alpha \diamond \gamma)$ in the prefix search tree have been discovered, it is unnecessary to search the branch under β. The reason is α and β share the exactly same descendants in the prefix search tree. So we can directly transplant the branch under α to β. The power of such transplanting is that only two operations

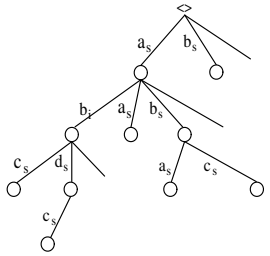
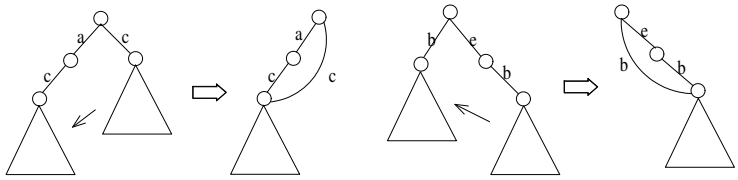


Fig. 1.3. Prefix Search Tree

needed to detect such condition: first, containment between α and β; second, comparison between $\mathcal{I}(S|_\alpha)$ and $\mathcal{I}(S|_\beta)$. Since $\mathcal{I}(S|_\alpha)$ is just a number and can be produced as a side-product when we project the database, the computation cost introduced by Lemma 2 is nearly negligible. We define *projected database closed set*, $LS = \{\alpha \mid support(s) \geqslant min\_support \text{ and } \nexists\beta, s.t.\ \alpha \sqsubseteq \beta \text{ and } \mathcal{I}(S|_\alpha) = \mathcal{I}(S|_\beta)\}$. Obviously, LS is a superset of closed frequent sequences. In CloSpan, instead of mining closed frequent sequences directly, it first produces the complete set of LS and then applies the non-closed sequence elimination in LS to generate the accurate set of closed frequent sequences.



(a) backward sub-pattern (b) backward super-pattern

Fig. 1.4. Backward Sub-Pattern and Super-Pattern

Corollary 1 (Backward Sub-Pattern). *If sequence α is discovered before β, $\alpha \sqsupset \beta$, and the condition $\mathcal{I}(S|_\alpha) = \mathcal{I}(S|_\beta)$ holds, it is sufficient to stop searching any descendant of β in the prefix search tree.*

We call β a *backward sub-pattern* of α if α is discovered before β and $\alpha \sqsupset \beta$. For the sample database in Table 1.1, if we know $\mathcal{I}(S|_{\langle ac\rangle}) = \mathcal{I}(S|_{\langle c\rangle})$, we can conclude that $S|_{\langle ac\rangle} = S|_{\langle c\rangle}$. We even need not compare the sequences in $S|_{\langle ac\rangle}$ and $S|_{\langle c\rangle}$ one by one to determine whether they are the same. This is the advantage of only comparing their size. Just as proved in Theorem 1, if their size is equal, we can conclude $S|_{\langle c\rangle} = S|_{\langle ac\rangle}$. We need not grow $\langle c\rangle$ anymore since all the children of $\langle c\rangle$ are the same as that of $\langle ac\rangle$ and vice versa under the condition of $S|_{\langle c\rangle} = S|_{\langle ac\rangle}$. Moreover, their supports

are the same. Therefore, any sequence beginning with $\langle c \rangle$ **is absorbed** by the sequences beginning with $\langle ac \rangle$. Figure 1.4(a) shows that their subtrees (descendant branches) can be merged into **one** without mining the subtree under $\langle c \rangle$.

Corollary 2 (Backward Super-Pattern). *If a sequence α is discovered before β, $\alpha \sqsubset \beta$, and the condition $\mathcal{I}(S|_\alpha) = \mathcal{I}(S|_\beta)$ holds, it is sufficient to transplanting the descendants of α to β instead of searching any descendant of β in the prefix search tree.*

We call β a *backward super-pattern* of α if α is discovered before β and $\alpha \sqsubset \beta$). For example, if we know $\mathcal{I}(S|_{\langle b \rangle}) = \mathcal{I}(S|_{\langle eb \rangle})$, we can conclude that $S|_{\langle eb \rangle} = S|_{\langle b \rangle}$. There is no need to grow $\langle eb \rangle$ since all the children of $\langle b \rangle$ are the same as those of $\langle eb \rangle$ and vice versa. Furthermore, they have the same support. Therefore, the sequences beginning with eb **can absorb** any sequence beginning with b. Figure 1.4(b) shows that their subtrees can be merged into **one** without discovering the subtree under $\langle eb \rangle$.

Based on the above discussion, we formulate the algorithm of CloSpan as follows.

Algorithm 2 (CloSpan) Closed frequent sequential pattern mining.

Input: A sequence database S, and the minimum support threshold $min\_support$.
Output: The candidate set of closed sequential patterns.
Method: Call CloSpan$(\langle\rangle, 0, S, L)$.

Subroutine CloSpan$(\alpha, l, S|_\alpha, L)$
The parameters are (1) α is a sequential pattern; (2) l is the length of α; and (3) $S|_\alpha$ is the α-projected database if $\alpha \neq \langle\rangle$, otherwise, it is the sequence database S. (4) L is the closed frequent sequence candidate set.
Method:

1. Check whether a discovered sequence β exists s.t. either $\alpha \sqsubseteq \beta$ or $\beta \sqsubseteq \alpha$, and $\mathcal{I}(S|_\alpha) = \mathcal{I}(S|_\beta)$; If such pattern exists, then apply Corollary 1 or 2 and return;
2. Insert α into L;
3. Scan $S|_\alpha$ once, find each frequent item, b, such that
   a) b can be assembled to the last element of α to form a sequential pattern; or
   b) $\langle b \rangle$ can be appended to α to form a sequential pattern;
4. For each frequent item b, append it to α to form a sequential pattern α';
5. For each α', if α' is frequent, construct α'-projected database $S|_{\alpha'}$, and call CloSpan$(\alpha', l+1, S|_{\alpha'}, L)$.

After we perform CloSpan$(\alpha, l, S|_\alpha, L)$, we get a closed frequent sequence candidate set, L. A post-processing step is required in order to delete non-closed sequential patterns existing in L.

1.5 Experimental Results and Performance Analysis

Since GSP [29] and SPADE [35] are the two most influential sequential pattern mining algorithms, we conduct an extensive performance study to compare PrefixSpan with them. In this section, we first report our experimental results on the performance of PrefixSpan in comparison with GSP and SPADE and then present our performance results of CloSpan in comparison with PrefixSpan.

1.5.1 Performance Comparison among PrefixSpan, FreeSpan, GSP and SPADE

To evaluate the effectiveness and efficiency of the PrefixSpan algorithm, we performed an extensive performance study of four algorithms: PrefixSpan, FreeSpan, GSP and SPADE, on both real and synthetic data sets, with various kinds of sizes and data distributions.

All experiments were conducted on a 750MHz AMD PC with 512 megabytes main memory, running Microsoft Windows-2000 Server. Three algorithms, GSP, FreeSpan, and PrefixSpan, were implemented by us using Microsoft Visual C++ 6.0. The implementation of the fourth algorithm, SPADE, is obtained directly from the author of the algorithm [35].

For the data sets used in our performance study, we use two kinds of data sets: one real data set and a group of synthetic data sets.

For real data set, we have obtained the *Gazelle* data set from Blue Martini. This data set has been used in KDD-CUP'2000 and contains totally 29369 customers' Web click-stream data provided by Blue Martini Software company. For each customer, there may be several sessions of web click-stream and each session can have multiple page views. Because each session is associated with both starting and ending date/time, for each customer we can sort its sessions of click-stream into a sequence of page views according to the viewing date/time. This dataset contains 29369 sequences (i.e., customers), 35722 sessions (i.e., transactions or events), and 87546 page views (i.e., products or items). There are in total 1423 distinct page views. More detailed information about this data set can be found in [16].

For synthetic data sets, we have also used a large set of synthetic sequence data generated by a data generator similar in spirit to the IBM data generator [2] designed for testing sequential pattern mining algorithms. Various kinds of sizes and data distributions of data sets are generated and tested in this performance study. The convention for the data sets is as follows: $C200T2.5S10I1.25$ means that the data set contains $200k$ customers (i.e., sequences) and the number of items is 10000. The average number of items in a transaction (i.e., event) is 2.5 and the average number of transactions in a sequence is 10. On average, a frequent sequential pattern consists of 4 transactions, and each transaction is composed of 1.25 items.

To make our experiments fair to all the algorithms, our synthetic test data sets are similar to that used in the performance study in [35]. Additional data sets are used for scalability study and for testing the algorithm behavior with varied (and sometimes very low) support thresholds.

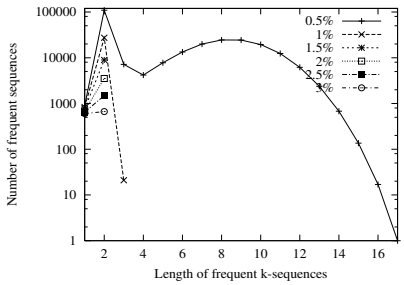


Fig. 1.5. Distribution of frequent sequences of data set $C10T8S8I8$.

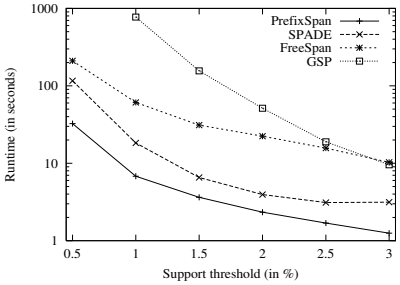


Fig. 1.6. Performance of the four algorithms on data set $C10T8S8I8$.

The first test of the four algorithms is on the data set $C10T8S8I8$, which contains $10k$ customers (i.e., sequences) and the number of items is 1000. Both the average number of items in a transaction (i.e., event) and the average number of transactions in a sequence are set to 8. On average, a frequent sequential pattern consists of 4 transactions, and each transaction is composed of 8 items. Figure 1.5 shows the distribution of frequent sequences of data set $C10T8S8I8$, from which one can see that when *min_support* is no less than 1%, the length of frequent sequences is very short (only 2-3), and the maximum number of frequent patterns in total is less than 10,000. Figure 1.6 shows the processing time of the four algorithms at different support thresholds. The processing times are sorted in time ascending order as "PrefixSpan $<$ SPADE $<$ FreeSpan $<$ GSP". When *min_support* $= 1\%$, PrefixSpan (runtime $= 6.8$ seconds) is about two orders of magnitude faster than GSP (runtime $=$ 772.72 seconds). When *min_support* is reduced to 0.5%, the data set contains a large number of frequent sequences, PrefixSpan takes 32.56 seconds, which is more than 3.5 times faster than SPADE (116.35 seconds), while GSP never terminates on our machine.

The performance study on the real data set *Gazelle* is reported as follows. Figure 1.7 shows the distribution of frequent sequences of Gazelle dataset for different support thresholds. We can see that this dataset is a very sparse dataset: only when the support threshold is lower than 0.05% are there some long frequent sequences. Figure 1.8 shows the performance comparison among the four algorithms for Gazelle dataset. From Figure 1.8 we can see that PrefixSpan is much more efficient than SPADE, FreeSpan and GSP. The SPADE algorithm is faster than both FreeSpan and GSP when the support threshold

is no less than 0.025%, but once the support threshold is no greater than 0.018%, it cannot stop running.

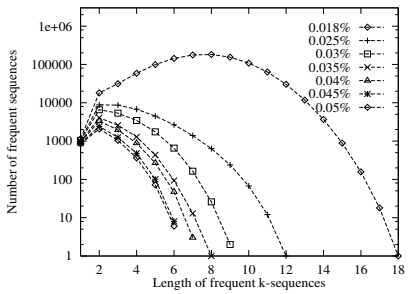


Fig. 1.7. Distribution of frequent sequences of data set Gazelle

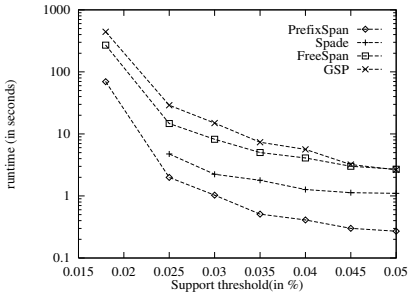


Fig. 1.8. Performance of the four algorithms on data set Gazelle

Finally, we compare the memory usage among the three algorithms, PrefixSpan, SPADE, and GSP using both real data set Gazelle and synthetic data set C200T5S10I2.5. Figure 1.9 shows the results for Gazelle dataset, from which we can see that PrefixSpan is efficient in memory usage. It consumes almost one order of magnitude less memory than both SPADE and GSP. For example, at support 0.018%, GSP consumes about 40 MB memory and SPADE just cannot stop running after it has used more than 22 MB memory while PrefixSpan only uses about 2.7 MB memory.

Figure 1.10 demonstrates the memory usage for dataset C200T5S10I2.5, from which we can see that PrefixSpan is not only more efficient but also more stable in memory usage than both SPADE and GSP. At support 0.25%, GSP cannot stop running after it has consumed about 362 MB memory and SPADE reported an error message "*memory:: Array: Not enough memory*" when it tried to allocate another bulk of memory after it has used about 262 MB memory, while PrefixSpan only uses 108 MB memory. This also explains why in several cases in our previous experiments when the support threshold becomes really low, only PrefixSpan can finish running.

Based on our analysis, PrefixSpan only needs memory space to hold the sequence datasets plus a set of header tables and pseudo-projection tables. Since the dataset C200T5S10I2.5 is about 46MB, which is much bigger than Gazelle (less than 1MB), it consumes more memory space than Gazelle but the memory usage is still quite stable (from 65 MB to 108 MB for different thresholds in our testing). However, both SPADE and GSP need memory space to hold candidate sequence patterns as well as the sequence datasets. When the *min_support* threshold drops, the set of candidate subsequences grows up quickly, which causes memory consumption upsurge, and sometimes both GSP and SPADE cannot finish processing.

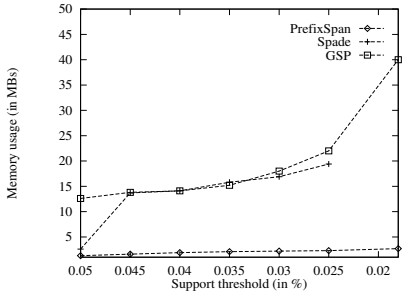


Fig. 1.9. Memory usage comparison among PrefixSpan, SPADE, and GSP for data set Gazelle

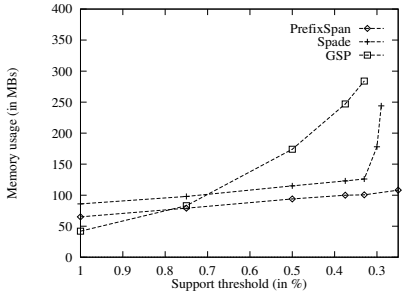


Fig. 1.10. Memory usage: PrefixSpan, SPADE, and GSP for synthetic data set C200T5S10I2.5

In summary, our performance study shows that PrefixSpan has the best overall performance among the four algorithms tested. SPADE, though weaker than PrefixSpan in most cases, outperforms GSP consistently, which is consistent with the performance study reported in [35]. GSP performs fairly well only when *min_support* is rather high, with good scalability, which is consistent with the performance study reported in [29]. However, when there are a large number of frequent sequences, its performance starts deteriorating. Our memory usage analysis also shows part of the reason why some algorithms becomes really slow because the huge number of candidate sets may consume a tremendous amount of memory. Also, when there are a large number of frequent subsequences, all the algorithms run slow. This problem can be partially solved by closed frequent sequential pattern mining. In the remaining of this section, we will demonstrate the compactness of closed patterns and the better performance achieved by CloSpan.

1.5.2 Performance comparison between CloSpan and PrefixSpan

The performance comparison between CloSpan and PrefixSpan are conducted in a different programming environment. All the experiments are done on a 1.7GHZ Intel Pentium-4 PC with 1GB main memory, running Windows XP Professional. All two algorithms are written in C++ with STL library support and compiled by g++ in cygwin environment with -O3 optimization.

The first experiment was conducted on the dataset $C10T2.5S6I2.5$ with 10k items. Figure 1.11 shows the run time of both PrefixSpan and CloSpan with different support threshold. PrefixSpan cannot complete the task below support threshold 0.001 due to too long runtime. Figure 1.12 shows the distribution of discovered frequent closed sequences in terms of length. With the decreasing minimum support, the maximum length of frequent closed sequences grows larger. Figure 1.13 shows the number of frequent sequences which are discovered and checked in order to generate the frequent closed sequence set. This number is roughly equal to how many times the procedure,

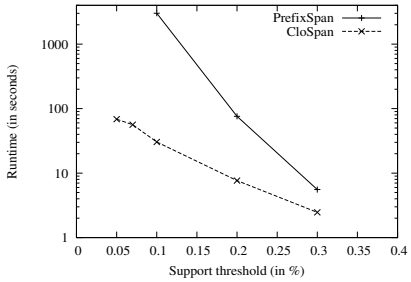


Fig. 1.11. The performance comparison between CloSpan and PrefixSpan on the dataset C10T2.5S6I2.5.

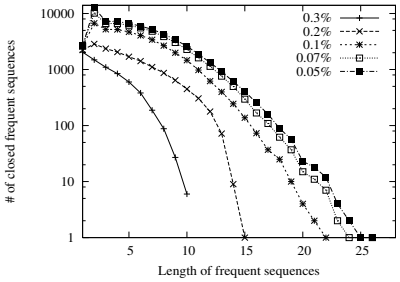


Fig. 1.12. The distribution of discovered frequent closed sequences in terms of length.

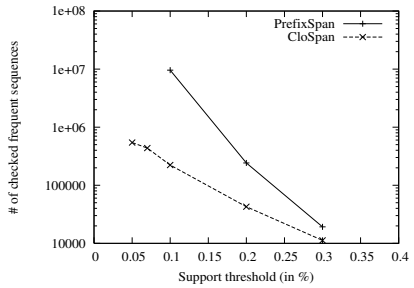


Fig. 1.13. The Number of frequent sequences checked.

CloSpan, is called and how many times projected databases are generated. Surprisingly, this number accurately predicates the total running time as the great similarity exists between Figure 1.11 and Figure 1.12. Therefore, for the same dataset, the number of checked frequent sequences approximately determines the performance.

We then test the performance of these two algorithms as some major parameters in the synthetic data generator are varied. The impact of different parameters is presented on the runtime of each algorithm. We select the following parameters as varied ones: the number of sequences in the dataset, the average number of transactions per sequence, and the average number of items per transaction. For each experiment, only one parameter varies with the others fixed. The experimental results are shown in Figures 1.14 to 1.16. We also discovered in other experiments, the speed-up decreases when the number of distinct items in the dataset goes down. However, it is still faster than PrefixSpan.

Direct mining of closed patterns leads to much fewer patterns, especially when the patterns are long or when the minimum support threshold is low.

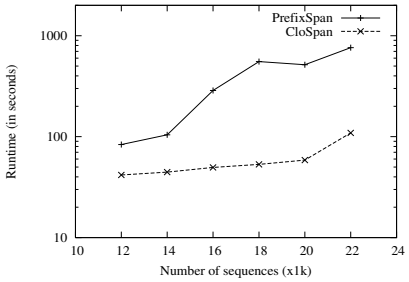


Fig. 1.14. Performance comparison vs. varying parameters: the number of sequences (C?T15S20I20, support threshold 0.8%.

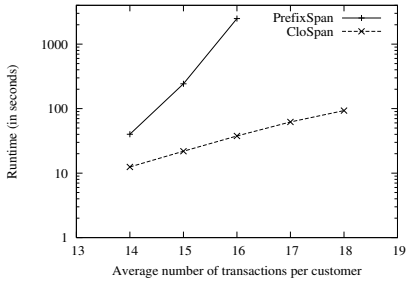


Fig. 1.15. Performance vs. varying parameters: Average number of transactions per sequence (C15T20S?I15, support threshold 0.75%).

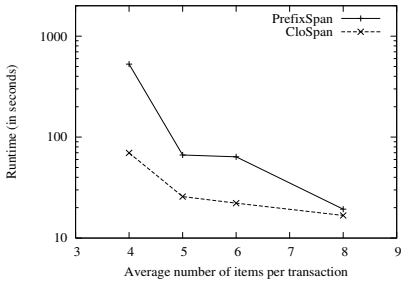


Fig. 1.16. Performance vs. varying parameters: Average number of items per transaction (C15T?S20I15, support threshold 1.2%).

According to our analysis, the sets of patterns derived from CloSpan have the same expressive power as the traditional sequential pattern mining algorithms. As indicated in Figure 1.13, CloSpan checks less frequent sequences than PrefixSpan and generates less number of projected databases. CloSpan clearly shows better performance than PrefixSpan in these cases. Our experimental results demonstrated this point since CloSpan often leads to savings in computation time over one order of magnitude in comparison with PrefixSpan. Based on the performance curves reported in Section 1.5.1 and the explosive number of subsequences generated for long sequences, it is expected that CloSpan will outperform GSP and SPADE as well when the patterns to be mined are long or when the support thresholds are low.

1.6 Extensions of Sequential Pattern Growth Approach

Comparing with mining (unordered) frequent patterns, mining sequential patterns is one step towards mining more sophisticated frequent patterns in large databases. With the successful development of sequential pattern-growth method, it is interesting to explore how such a method can be extended to handle more sophisticated mining requests. In this section, we will discuss a few extensions of the sequential pattern growth approach.

1.6.1 Mining multi-dimensional, multi-level sequential patterns

In many applications, sequences are often associated with different circumstances, and such circumstances form a multiple dimensional space. For example, customer purchase sequences are associated with region, time, customer group, and others. It is interesting and useful to mine sequential patterns associated with *multi-dimensional information*. For example, one may find that retired customers (with age) over 60 may have very different patterns in shopping sequences from the professional customers younger than 40. Similarly, items in the sequences may also be associated with *different levels of abstraction*, and such multiple abstraction levels will form a multi-level space for sequential pattern mining. For example, one may not be able to find any interesting buying patterns in an electronics store by examining the concrete models of products that customers purchase. However, if the concept level is raised a little high to brand-level, one may find some interesting patterns, such as "*if one bought an IBM PC, it is likely s/he will buy a new IBM Laptop and then a Cannon digital camera within the next six months.*"

There have been numerous studies at mining frequent patterns or associations at multiple levels of abstraction, such as [2, 9], and mining association or correlations at multiple dimensional space, such as [15, 6]. One may like to see how to extend the framework to mining sequential patterns in multi-dimensional, multi-level spaces.

Interestingly, pattern growth-based methods, such as PrefixSpan, can be naturally extended to mining such patterns. Here is an example illustrating one such extension.

Example 8 (Mining multi-dimensional, multi-level sequential patterns). Consider a sequence database SDB in Table 1.4, where each sequence is associated with certain multi-dimensional, multi-level information. For example, it may contain multi-dimensional circumstance information, such as *cust-grp* = *business*, *city* = *Boston*, and *age-grp* = *middle_aged*. Also, each item may be associated with multiple-level information, such as item b being *IBM Laptop Thinkpad_X30*.

PrefixSpan can be extended to mining sequential patterns efficiently in such a multi-dimensional, multi-level environment. One such solution which we call *uniform sequential* (or *Uni-Seq*) [27] is outlined as follows. For each sequence, a set of multi-dimensional circumstance values can be treated as one

cid	cust-grp	city	age-grp	sequence
10	business	Boston	middle_aged	$\langle(bd)cba\rangle$
20	professional	Chicago	young	$\langle(bf)(ce)(fg)\rangle$
30	business	Chicago	middle_aged	$\langle(ah)abf\rangle$
40	education	New York	retired	$\langle(be)(ce)\rangle$

Table 1.4. A multi-dimensional sequence database

added transaction in the sequence. For example, for $cid = 10$, (*business*, *Boston*, *middle_aged*) can be added into the sequence as one additional transaction. Similarly, for each item b, its associated multi-level information can be added as additional items into the same transaction that b resides. Thus the first sequence can be transformed into a sequence cid_{10} as, $cid_{10} : \langle$ (*business*, *Boston*, *middle_aged*), $((IBM, Laptop, Thinkpad\_X30)$, $(Dell, PC, Precision\_330))$ $(Canon, digital\_camera, CD420)$, $(IBM$, *Laptop*, $Thinkpad\_X30)$, $(Microsoft, RDBMS, SQLServer\_2000)\rangle$. With such transformation, the database becomes a typical single-dimensional, single-level sequence database, and the PrefixSpan algorithm can be applied to efficient mining of multi-dimensional, multi-level sequential patterns. □

The proposed embedding of multi-dimensional, multi-level information into a transformed sequence database, and then extension of PrefixSpan to mining sequential patterns, as shown in Example 8, has been studied and implemented in [27]. In the study, we propose a few alternative methods, which integrate some efficient cubing algorithms, such as BUC [5] and H-cubing [10], with PrefixSpan. A detailed performance study in [27] shows that the *Uni-Seq* is an efficient algorithm. Another interesting algorithm, called *Seq-Dim*, which first mines sequential patterns, and then for each sequential pattern, forms projected multi-dimensional database and finds multi-dimensional patterns within the projected databases, also shows high performance in some situations. In both cases, PrefixSpan forms the kernel of the algorithm for efficient mining of multi-dimensional, multi-level sequential patterns.

1.6.2 Constraint-based mining of sequential patterns

For many sequential pattern mining applications, instead of finding all the possible sequential patterns in a database, a user may often like to enforce certain constraints to find desired patterns. The mining process which incorporates user-specified constraints to reduce search space and derive only the user-interested patterns is called *constraint-based mining*.

Constraint-based mining has been studied extensively in frequent pattern mining, such as [22, 3, 24]. In general, constraints can be characterized based on the notion of monotonicity, anti-monotonicity, succinctness, as well as convertible and inconvertible constraints respectively, depending on whether a

constraint can be transformed into one of these categories if it does not naturally belong to one of them [24]. This has become a classical framework for constraint-based frequent pattern mining.

Interestingly, such a constraint-based mining framework can be extended to sequential pattern mining. Moreover, with pattern-growth framework, some previously not-so-easy-to-push constraints, such as regular expression constraints [7] can be handled elegantly. Let's examine one such example.

Example 9 (Constraint-based sequential pattern mining). Suppose our task is to mine sequential patterns with a regular expression constraint $C = \langle a * \{bb|(bc)d|dd\}\rangle$ with $min\_support = 2$, in a sequence database S (Table 1.1).

Since a regular expression constraint, like C, is neither anti-monotone, nor monotone, nor succinct, the classical constraint-pushing framework [22] cannot push it deep. To overcome this difficulty, Garofalakis, et al. [7] develop 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. However, the basic evaluation framework for sequential patterns is still based on GSP [29], a typical candidate generation-and-test approach.

With the development of the pattern-growth methodology, such kinds of constraints can be pushed deep easily and elegantly into the sequential pattern mining process [26]. This is because in the context of PrefixSpan a regular expression constraint has a nice property called *growth-based anti-monotonic*. A constraint is *growth-based anti-monotonic* if it has the following property: *if a sequence α satisfies the constraint, α must be reachable by growing from any component which matches part of the regular expression.*

The constraint $C = \langle a * \{bb|(bc)d|dd\}\rangle$ can be integrated with the pattern-growth mining process as follows. First, only the $\langle a\rangle$-projected database needs to be mined since the regular expression constraint C starting with a, and only the sequences which contain frequent single item within the set of $\{b, c, d\}$ should retain in the $\langle a\rangle$-projected database. Second, the remaining mining can proceed from the suffix, which is essentially "*Suffix-Span*", an algorithm symmetric to PrefixSpan by growing suffixes from the end of the sequence forward. The growth should match the suffix constraint "$\langle\{bb|(bc)d|dd\}\rangle$". For the projected databases which matches these suffixes, one can grow sequential patterns either in prefix- or suffix- expansion manner to find all the remaining sequential patterns. □

Notice that the regular expression constraint C given in Example 9 is in a special form "$\langle prefix * suffix\rangle$" out of many possible general regular expressions. In this special case, an integration of PrefixSpan and *Suffix-Span* may achieve the best performance. In general, a regular expression could be of the form "$\langle * \alpha_1 * \alpha_2 * \alpha_3 * \rangle$", where α_i is a set of instantiated regular expressions. In this case, FreeSpan should be applied to push the instantiated items by expansion first from the instantiated items. A detailed discussion of constraint-based sequential pattern mining is in [26].

1.6.3 Mining top-k closed sequential patterns

Mining closed patterns may significantly reduce the number of patterns generated and is *information lossless* because it can be used to derive the complete set of sequential patterns. However, setting min_support is a subtle task: *A too small value may lead to the generation of thousands of patterns, whereas a too big one may lead to no answer found.* To come up with an appropriate min_support, one needs prior knowledge about the mining query and the task-specific data, and be able to estimate beforehand how many patterns will be generated with a particular threshold.

As proposed in [13], a desirable solution is to change the task of mining frequent patterns to *mining top-k frequent closed patterns of minimum length* min_ℓ, where k is the number of closed patterns to be mined, top-k refers to the k most frequent patterns, and min_ℓ is the minimum length of the closed patterns. We develop TSP [30] to discover top-k closed sequences. TSP is a multi-pass search space traversal algorithm that finds the most frequent patterns early in the mining process and allows dynamic raising of min_support which is then used to prune unpromising branches in the search space. Also, TSP devises an efficient closed pattern verification method which guarantees that during the mining process the candidate result set consists of the desired number of closed sequential patterns. The efficiency of TSP is further improved by applying the minimum length constraint in the mining and by employing the early termination conditions developed in CloSpan [34].

1.6.4 Mining approximate consensus sequential patterns

As we discussed before, conventional sequential pattern mining methods may meet inherent difficulties in mining databases with long sequences and noise. They may generate a huge number of short and trivial patterns but fail to find interesting patterns approximately shared by many sequences. In many applications, it is necessary to mine sequential patterns approximately shared by many sequences.

To attack these problems, in [17], we propose the theme of *approximate sequential pattern mining* roughly defined as *identifying patterns approximately shared by many sequences.* We present an efficient and effective algorithm, *ApproxMap* (for APPROXimate Multiple Alignment Pattern mining), to mine consensus patterns from large sequence databases. The method works in two steps. First, the sequences are clustered by similarity. Then, the consensus patterns are mined directly from each cluster through multiple alignments. A novel structure called weighted sequence is used to compress the alignment result. For each cluster, the longest consensus pattern best representing the cluster is generated from its weighted sequence.

Our extensive experimental results on both synthetic and real data sets show that *ApproxMap* is robust to noise and is both effective and efficient in mining approximate sequential patterns from noisy sequence databases with

lengthy sequences. In particular, we report a successful case of mining a real data set which triggered important investigations in welfare services.

1.6.5 Clustering time series gene expression

Clustering the time series gene expression data is an important task in bioinformatics research and biomedical applications. Time series gene expression data is also in the form of sequences. Recently, some clustering methods have been adapted or proposed. However, some problems still remain, such as the robustness of the mining methods, the quality and the interpretability of the mining results.

In [14], we tackle the problem of *effectively clustering time series gene expression data* by proposing algorithm *DHC*, a *density-based, hierarchical clustering method aiming at time series gene expression data*. We use a density-based approach to identifying the clusters such that the clustering results are with high quality and robustness. Moreover, The mining result is in the form of a *density tree*, which uncovers the embedded clusters in a data sets. The inner-structures, the borders and the outliers of the clusters can be further investigated using the *attraction tree*, which is a intermediate result of the mining. By these two trees, the internal structure of the data set can be visualized effectively. Our empirical evaluation using some real-world data sets show that the method is effective, robust and scalable. It matches the ground truth given by bioinformatics experts very well in the sample data sets.

1.6.6 Towards mining more complex kinds of structured patterns

Besides mining sequential patterns, another important task is the mining of frequent sub-structures in a database composed of structured or semi-structured data sets. The substructures may consist of trees, directed-acyclic graphs (i.e., DAGs), or general graphs which may contain cycles. There are a lot of applications related to mining frequent substructures since most human activities and natural processes may contain certain structures, and a huge amount of such data has been collected in large data/information repositories, such as molecule or bio-chemical structures, Web connection structures, and so on. It is important to develop scalable and flexible methods for mining structured patterns in such databases. There have been some recent work on mining frequent subtrees, such as [37], and frequent subgraphs, such as [18, 32] in structured databases, where [32] shows that the pattern growth approach has clear performance edge over a candidate generation-and-test approach. Furthermore, as discussed above, is it is more desirable to mine closed frequent subgraphs (a subgraph g is *closed* if there exists no super-graph of g carrying the same support as g) than mining explicitly the complete set of frequent subgraphs because a large graph inherently contains an exponential number of subgraphs. A recent study [33] has developed an efficient closed subgraph

pattern method, called *CloseGraph*, which is also based on the pattern-growth framework and influenced by this approach.

1.7 Conclusions

We have introduced a *pattern-growth approach* for efficient and scalable mining of sequential patterns in large sequence databases. Instead of refinement of the Apriori-like, candidate generation-and-test approach, such as GSP [29] and SPADE[35], we promote a divide-and-conquer approach, called *pattern-growth approach*, which is an extension of FP-growth [12], an efficient pattern-growth algorithm for mining frequent patterns without candidate generation.

An efficient pattern-growth method is developed for mining frequent sequential patterns, represented by PrefixSpan, and mining closed sequential patterns, represented by CloSpan, are presented and studied in this paper.

PrefixSpan recursively projects a sequence database into a set of smaller projected sequence databases and grows sequential patterns in each projected database by exploring only locally frequent fragments. It mines the complete set of sequential patterns and substantially reduces the efforts of candidate subsequence generation. Since PrefixSpan explores ordered growth by prefix-ordered expansion, it results in less "growth points" and reduced projected databases in comparison with our previously proposed pattern-growth algorithm, FreeSpan. Furthermore, a *pseudo-projection* technique is proposed for PrefixSpan to reduce the number of physical projected databases to be generated.

CloSpan mines closed sequential patterns efficiently by discovery of sharing portions of the projected databases in the mining process and prune any redundant search space and therefore substantially enhanced the mining efficiency and reduces the redundant patterns.

Our comprehensive performance study shows that PrefixSpan outperforms the Apriori-based GSP algorithm, FreeSpan, and SPADE in most cases, and PrefixSpan integrated with pseudo-projection is the fastest among all the tested algorithms for mining the complete set of sequential patterns; whereas CloSpan may substantially improve the mining efficiency over PrefixSpan and returns a substantially smaller set of results while preserving the completeness of the answer sets.

Based on our view, the implication of this method is far beyond yet another efficient sequential pattern mining algorithm. It demonstrates the strength of the pattern-growth mining methodology since the methodology has achieved high performance in both frequent-pattern mining and sequential pattern mining. Moreover, our discussion shows that the methodology can be extended to mining multi-level, multi-dimensional sequential patterns, mining sequential patterns with user-specified constraints, and a few interesting applications. Therefore, it represents a promising approach for the applications that rely on the discovery of frequent patterns and/or sequential patterns.

There are many interesting issues that need to be studied further. Especially, the developments of specialized sequential pattern mining methods for particular applications, such as DNA sequence mining that may admit faults, such as allowing insertions, deletions and mutations in DNA sequences, and handling industry/engineering sequential process analysis are interesting issues for future research.

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. R. J. Bayardo, R. Agrawal, and D. Gunopulos. Constraint-based rule mining on large, dense data sets. In *Proc. 1999 Int. Conf. Data Engineering (ICDE'99)*, pages 188–197, Sydney, Australia, April 1999.
4. C. Bettini, X. S. Wang, and S. Jajodia. Mining temporal relationships with multiple granularities in time sequences. *Data Engineering Bulletin*, 21:32–38, 1998.
5. K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *Proc. 1999 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'99)*, pages 359–370, Philadelphia, PA, June 1999.
6. G. Grahne, L. V. S. Lakshmanan, X. Wang, and M. H. Xie. On dual mining: From patterns to circumstances, and back. In *Proc. 2001 Int. Conf. Data Engineering (ICDE'01)*, pages 195–204, Heidelberg, Germany, April 2001.
7. S. Guha, R. Rastogi, and K. Shim. Rock: A robust clustering algorithm for categorical attributes. In *Proc. 1999 Int. Conf. Data Engineering (ICDE'99)*, pages 512–521, Sydney, Australia, Mar. 1999.
8. 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.
9. J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proc. 1995 Int. Conf. Very Large Data Bases (VLDB'95)*, pages 420–431, Zurich, Switzerland, Sept. 1995.
10. J. Han, J. Pei, G. Dong, and K. Wang. Efficient computation of iceberg cubes with complex measures. In *Proc. 2001 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'01)*, pages 1–12, Santa Barbara, CA, May 2001.
11. 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 ACM SIGKDD Int. Conf. Knowledge Discovery in Databases (KDD'00)*, pages 355–359, Boston, MA, Aug. 2000.
12. 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.
13. J. Han, J. Wang, Y. Lu, and P. Tzvetkov. Mining top-k frequent closed patterns without minimum support. In *Proc. 2002 Int. Conf. on Data Mining (ICDM'02)*, pages 211–218, Maebashi, Japan, Dec. 2002.

14. D. Jiang, J. Pei, and A. Zhang. DHC: A density-based hierarchical clustering method for gene expression data. In *Proc. 3rd IEEE Symp. Bio-informatics and Bio-engineering (BIB'03)*, Washington D.C., March 2003.
15. M. Kamber, J. Han, and J. Y. Chiang. Metarule-guided mining of multi-dimensional association rules using data cubes. In *Proc. 1997 Int. Conf. Knowledge Discovery and Data Mining (KDD'97)*, pages 207–210, Newport Beach, CA, Aug. 1997.
16. R. Kohavi, C. Brodley, B. Frasca, L. Mason, and Z. Zheng. KDD-Cup 2000 organizers' report: Peeling the onion. *SIGKDD Explorations*, 2:86–98, 2000.
17. H. Kum, J. Pei, and W. Wang. Approxmap: Approximate mining of consensus sequential patterns. In *Proc. 2003 SIAM Int. Conf. on Data Mining (SDM '03)*, San Francisco, CA, May 2003.
18. M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proc. 2001 Int. Conf. Data Mining (ICDM'01)*, pages 313–320, San Jose, CA, Nov. 2001.
19. 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.
20. H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1:259–289, 1997.
21. F. Masseglia, F. Cathala, and P. Poncelet. The psp approach for mining sequential patterns. In *Proc. 1998 European Symp. Principle of Data Mining and Knowledge Discovery (PKDD'98)*, pages 176–184, Nantes, France, Sept. 1998.
22. R. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proc. 1998 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'98)*, pages 13–24, Seattle, WA, June 1998.
23. 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.
24. J. Pei, J. Han, and L. V. S. Lakshmanan. Mining frequent itemsets with convertible constraints. In *Proc. 2001 Int. Conf. Data Engineering (ICDE'01)*, pages 433–332, Heidelberg, Germany, April 2001.
25. 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.
26. J. Pei, J. Han, and W. Wang. Constraint-based sequential pattern mining in large databases. In *Proc. 2002 Int. Conf. Information and Knowledge Management (CIKM'02)*, pages 18–25, McLean, VA, Nov. 2002.
27. H. Pinto, J. Han, J. Pei, K. Wang, Q. Chen, and U. Dayal. Multi-dimensional sequential pattern mining. In *Proc. 2001 Int. Conf. Information and Knowledge Management (CIKM'01)*, pages 81–88, Atlanta, GA, Nov. 2001.
28. S. Ramaswamy, S. Mahajan, and A. Silberschatz. On the discovery of interesting patterns in association rules. In *Proc. 1998 Int. Conf. Very Large Data Bases (VLDB'98)*, pages 368–379, New York, NY, Aug. 1998.
29. 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.

30. P. Tzvetkov, X. Yan, and J. Han. Tsp: Mining top-k closed sequential patterns. In *Proc. 2003 Int. Conf. Data Mining (ICDM'03)*, Melbourne, FL, Nov. 2003.
31. J. Wang, G. Chirn, T. Marr, B. Shapiro, D. Shasha, and K. Zhang. Combinatorial pattern discovery for scientific data: Some preliminary results. In *Proc. 1994 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'94)*, pages 115–125, Minneapolis, MN, May, 1994.
32. X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *Proc. 2002 Int. Conf. on Data Mining (ICDM'02)*, pages 721–724, Maebashi, Japan, Dec. 2002.
33. X. Yan and J. Han. CloseGraph: Mining closed frequent graph patterns. In *Proc. 2003 ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (KDD'03)*, Washington, D.C., Aug. 2003.
34. X. Yan, J. Han, and R. Afshar. CloSpan: Mining closed sequential patterns in large datasets. In *Proc. 2003 SIAM Int. Conf. Data Mining (SDM'03)*, pages 166–177, San Fransisco, CA, May 2003.
35. M. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning*, 40:31–60, 2001.
36. M. J. Zaki. Efficient enumeration of frequent sequences. In *Proc. 7th Int. Conf. Information and Knowledge Management (CIKM'98)*, pages 68–75, Washington DC, Nov. 1998.
37. M. J. Zaki. Efficiently mining frequent trees in a forest. In *Proc. 2002 ACM SIGKDD Int. Conf. Knowledge Discovery in Databases (KDD'02)*, pages 71–80, Edmonton, Canada, July 2002.
38. M. J. Zaki and C. J. Hsiao. CHARM: An efficient algorithm for closed itemset mining. In *Proc. 2002 SIAM Int. Conf. Data Mining (SDM'02)*, pages 457–473, Arlington, VA, April 2002.