May 2004, Vol.19, No.3, pp.257–279 J. Comput. Sci. & Technol.

From Sequential Pattern Mining to Structured Pattern Mining: A Pattern-Growth Approach

Jia-Wei Han[1], Jian Pei[2], and Xi-Feng Yan[1]

[1] *University of Illinois at Urbana-Champaign, Urbana, IL 61801, U.S.A.*

[2] *State University of New York at Buffalo, Buffalo, NY 14260-2000, U.S.A.*

E-mail: {hanj, xyan}@cs.uiuc.edu; jianpei@cse.buffalo.edu

Received January 19, 2004.

Abstract 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, a horizontal format-based sequential pattern mining method, and (ii) SPADE, a vertical format-based method; and (2) a *pattern-growth* method, represented by PrefixSpan and its further extensions, such as gSpan for mining structured patterns.

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 and PrefixSpan, for efficient sequential pattern mining. Then we introduce gSpan for mining structured patterns using the same methodology. Their relative performance in large databases is presented and analyzed. Several extensions of these methods are also discussed in the paper, including mining multi-level, multi-dimensional patterns and mining constraint-based patterns.

Keywords data mining, sequential pattern mining, structured pattern mining, scalability, performance analysis

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 [1] 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.*

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 $(x_1 x_2 \ldots x_m)$, 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 $1 \leqslant j_1 < j_2 < \cdots < j_n \leqslant m$ 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., $support_S(\alpha) = |\{\langle sid, s \rangle | (\langle sid, s \rangle \in S) \wedge (\alpha \sqsubseteq s)\}|$. It can be denoted by $support(\alpha)$ if the sequence database is clear from the context. Given a positive

*Survey

The work was supported in part by the Natural Sciences and Engineering Research Council of Canada, the Networks of Centres of Excellence of Canada, the Hewlett-Packard Lab, the U.S. National Science Foundation (Grant Nos. 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.

integer *min_support* as the **support threshold**, a sequence α is called a **sequential pattern** in sequence database S if $support_S(\alpha) \geqslant min\_support$. A sequential pattern with length l is called an l-**pattern**.

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

Table 1. A Sequence Database

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	$\langle eg(af)cbc\rangle$

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 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 1 and 3 *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[1−11]. Srikant and Agrawal[2] generalized their definition of sequential patterns in [1] 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.*[3] presented a problem of mining frequent episodes in a sequence of events, where episodes are essentially acyclic graphs of events whose edges specify the temporal precedent-subsequent relationship without restriction on interval. Bettini, *et al.*[5] 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.*[8] proposed inter-transaction association rules that are implication rules whose two sides are totally-ordered episodes with timing-interval restrictions. Garofalakis, *et al.*[12] 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.*[9] 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.*[10] developed a frequent pattern mining method for mining partial periodic 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[13] 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 [6, 14].

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 fre-*

quent, and based on a candidate generation-and-test paradigm proposed in association mining[15].

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**quent pattern-projected **S**equential **pa**ttern mining)[16], 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[13] (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.

The pattern-growth methodology exposed in PrefixSpan can also be used for structured pattern mining: *finding frequent subgraphs over a collection of graphs.* Frequent subgraphs are common topological structures buried in a graph dataset. For example, in chemical informatics, we can view each chemical compound as a graph which consists of atoms and bonds. The AIDS antiviral screen data provided by Developmental Theroapeutics Program in NCI/NIH tells us, among the chemical compounds tested, which one is active to inhibit HIV virus, and which one is not. By analyzing the small chemical structures embedded in each compound, we find that some structures appear more commonly in the active dataset but rarely in the inactive dataset. This kind of knowledge will deepen our understanding of these structures and improve our ability in drug design.

We develop an efficient structured pattern mining algorithm, gSpan (*mining frequent graph patterns by subgraph Spanning*). It was inspired by PrefixSpan: It grows patterns from a single pattern directly. gSpan extends the pattern-growth methodology to combine graph pattern growing and frequency counting into one procedure. Thus, it avoids the significant overhead introduced by candidate generation. The Apriori-like approach has to use breadth-first search (BFS) strategy because of its intrinsic level-wise candidate generation: In order to know whether a k-edge graph is frequent or not, it has to check all of its $(k-1)$-edge subgraphs to get the upper bound of its frequency. Thus, before mining any k-edge subgraph, the Apriori-like approach must finish mining $(k-1)$-edge subgraphs. Therefore, BFS is necessary in the Apriori-like approach. In contrast, the pattern-growth approach is more flexible on the search method: Both breadth-first search and depth-first search (DFS) can work. Usually BFS has higher memory consumption than DFS.

In the following sections, we will systematically explore how to mine sequential patterns and structured patterns.

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[15] to sequential pattern analysis. Similar to frequent patterns, sequential patterns has the anti-monotone (i.e., downward closure) property as follows: *every non-empty subsequence 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[2], and (2) a vertical data format based sequential pattern mining method: SPADE[13]. We outline and analyze these two methods in this section.

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[2] by extension of their frequent itemset mining algorithm, Apriori[15]. 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 $6 \times 6 + \frac{6\times 5}{2} = 51$ candidate sequences, $C_2 = \{\langle aa\rangle, \langle ab\rangle, \ldots, \langle af\rangle, \langle ba\rangle, \langle bb\rangle, \ldots, \langle ff\rangle, \langle (ab)\rangle, \langle (ac)\rangle, \ldots, \langle (ef)\rangle\}$.

The multi-scan mining process is shown in Fig.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-$(k-1)$ 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.

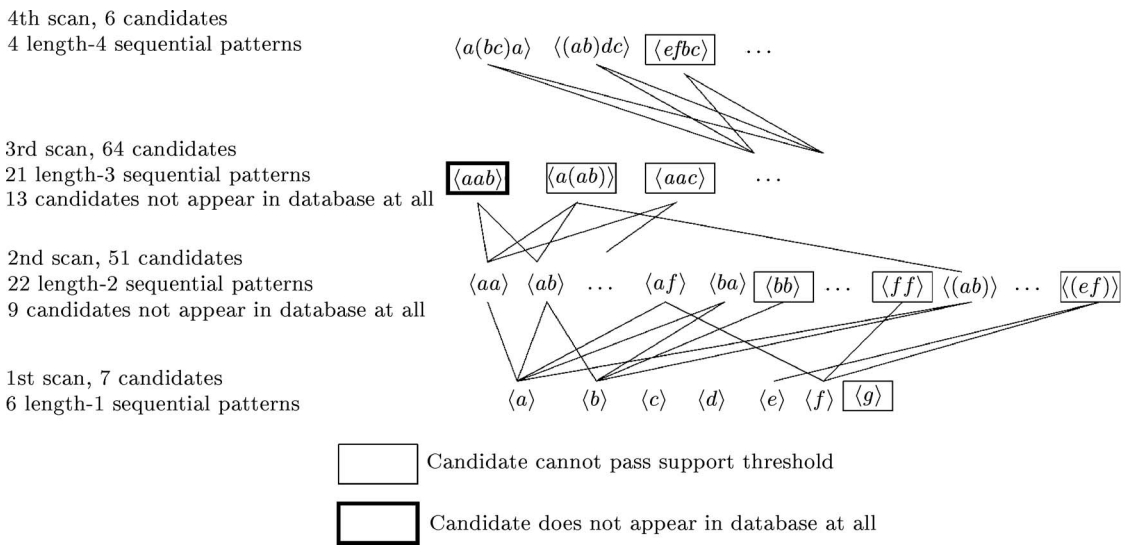

Fig.1. Candidates, candidate generation, and sequential patterns in GSP.

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 $1,000$ frequent sequences of length-1, such as $\langle a_1\rangle$, $\langle a_2\rangle, \ldots, \langle a_{1,000}\rangle$, an Apriori-like algorithm will generate $1,000 \times 1,000 + \frac{1,000\times 999}{2} = 1,499,500$ candidate sequences, where the first term is derived from the set $\langle a_1a_1\rangle$, $\langle a_1a_2\rangle, \ldots, \langle a_1a_{1,000}\rangle$, $\langle a_2a_1\rangle$, $\langle a_2a_2\rangle, \ldots, \langle a_{1,000}a_{1,000}\rangle$, and the second term is derived from the set $\langle (a_1a_2)\rangle$, $\langle (a_1a_3)\rangle, \ldots, \langle (a_{999}a_{1,000})\rangle$.

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, $\langle a_1a_2\ldots a_{100}\rangle$, 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 $\sum_{i=1}^{100}\binom{100}{i} = 2^{100}-1 \approx 10^{30}$.

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*?"

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 [13], 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 Fig.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 continues until no frequent sequences can be found or no such sequences can be formed

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

by such joins.

Some fragments of the SPADE mining process are illustrated in Fig.2. The detailed analysis of the method can be found in [13].

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.

2.3 FSG: An Apriori-Based Structured Pattern Mining Algorithm

FSG is an Apriori-based algorithm for finding all connected subgraphs that appear frequently in a graph database. It uses the same candidate generation-and-test strategy adopted by Apriori. FSG requires a join operation which merges two k-edge frequent subgraphs in order to generate a $(k+1)$-edge candidate subgraph. If the candidate subgraph is frequent, then all of its subgraphs must be frequent. By checking this downward closure property, FSG may prune lots of infrequent candidate subgraphs. FSG is a typical Apriori-based approach. There are other algorithms using the same strategy but different pattern expansion blocks: vertices in [17], edges in [18], and edge-disjoint paths in [19]. In the context of frequent graph mining, Apriori-like algorithms suffer from two kinds of considerable overheads: (1) joining two k-edge frequent graphs to generate $(k+1)$-edge graph candidates, and (2) checking the frequency of these candidates separately. These two operations usually are the performance bottlenecks of Apriori-like algorithms.

3 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[20]. We introduce first the FreeSpan algorithm and then a more efficient alternative, the PrefixSpan algorithm.

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 subdatabases 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 ($1 \leqslant i \leqslant n$) 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-1 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 [16]. 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 nontrivial 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.

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 databases. 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 $\beta = \langle e'_1 e'_2 \cdots e'_m\rangle$ $(m \leqslant n)$ is called a **prefix** of α if and only if (1) $e'_i = e_i$ for $(i \leqslant m-1)$; (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 $e''_m = (e_m - e'_m)$. [①] 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 $\langle a(bc)\rangle$ 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.

• 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 $(1 \leqslant i \leqslant n)$ is the set of sequential patterns with prefix $\langle x_i\rangle$.

• 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 $(1 \leqslant j \leqslant m)$ 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 the sequence resulted from the 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 $\gamma/\beta = (\gamma/\alpha)/\beta$.

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

The second part of the lemma states that to collect the 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 on our running example.

Example 6 (PrefixSpan). For the same sequence database S in Table 1 with $min\_sup = 2$, sequential patterns in S can be mined by a prefix-projection method in the following steps.

Step 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.

Step 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 $\langle b\rangle, \ldots$, and (6) the ones with prefix $\langle f\rangle$.

Step 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 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.

Table 2. Projected Databases and Sequential Patterns

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

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 $\langle ab\rangle, \ldots$, 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 re-

turns 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 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 2.

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

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 α'; or
   b) $\langle b\rangle$ can be appended to α to form a sequential pattern α'.
2. For each α', if α' is frequent, output α', 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. Compared 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.

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 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 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 sequence_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 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 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.

4 Pattern-Growth Approach for Structured Pattern Mining

In this section, we introduce a structured pattern mining algorithm using the pattern-growth methodology, which has a strong relation with the sequential pattern mining algorithms presented in Section 3.

As a general data structure, labeled graph can be used to model structured patterns among data. A *labeled graph* has labels associated with its edges and vertices. We denote the vertex set of a graph g by $V(g)$, the edge set by $E(g)$. A label function, l, can map a vertex or an edge to a label. A graph g is a subgraph of another graph g' if there exists a subgraph isomorphism from g to g'.

Definition 1 (Subgraph Isomorphism). *A subgraph isomorphism is an injective function f: $V(g) \to V(g')$, such that* (1) $\forall u \in V(g)$, $l(u) = l'(f(u))$, *and* (2) $\forall (u, v) \in E(g)$, $(f(u), f(v)) \in E(g')$ *and* $l(u, v) = l'(f(u), f(v))$, *where l and l' are the label functions of g and g' respectively.*

If g is a *subgraph* of g', then g' is a *supergraph* of g, denoted by $g \subseteq g'$ (*proper supergraph*, if $g \subset g'$). Given a labeled graph dataset, $S = \{G_1, G_2, \ldots, G_n\}$, $support(g)$ (or $frequency(g)$) denotes the percentage (or number) of graphs (in S) in which g is a subgraph. The set of **frequent**

Table 3. A Sequence Database and Some of Its Pseudo-Projected Databases

Sequence_id	Sequence	$\langle a\rangle$	$\langle b\rangle$	$\langle c\rangle$	$\langle d\rangle$	$\langle f\rangle$	$\langle aa\rangle$	...
1	$\langle a(abc)(ac)d(cf)\rangle$	2, 3, 6	4	5, 7	8	\$	3, 6	...
2	$\langle (ad)c(bc)(ae)\rangle$	2	5	4, 6	3	$\emptyset$	7	...
3	$\langle (ef)(ab)(df)cb\rangle$	4	5	8	6	3, 7	$\emptyset$	...
4	$\langle eg(af)cbc\rangle$	4	6	6	$\emptyset$	5	$\emptyset$	...

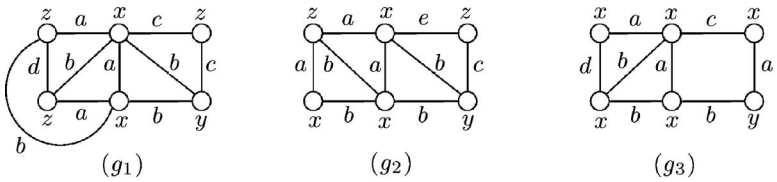

Fig.3. Sample graph dataset S.

graph patterns, F, includes all the graphs whose support is no less than a minimum support threshold, *min_support*.

Since most of interesting graph patterns are connected graphs, we first study mining labeled connected undirected graphs without multiple edges. Our method can be easily extended for mining other kinds of graph structures such as unlabeled graphs, graphs with self-loops and multiple edges, directed graphs, disconnected graphs, and so on.

Example 8. Fig.3 is a sample labeled graph dataset, S, where three labeled graphs are presented. Among these graphs, each vertex and edge are assigned a label. Let min_sup be 2. The alphabetic order is taken as the default lexicographical order.

A graph g can be extended by adding a new edge e. Let the new graph be denoted by $g \diamond_x e$. Edge e may or may not introduce a new vertex to g. If e introduces a new vertex, we denote the new graph by $g \diamond_{xf} e$, otherwise, $g \diamond_{xb} e$. Algorithm 2 illustrates a naive frequent graph mining algorithm. It finds all the frequent graphs, closed or non-closed. For each discovered graph g, it performs the extension recursively until all the frequent graphs with g embedded are discovered.

Algorithm 2 (NaiveGraph). Naive structured pattern mining.

Input: A graph database S, the frequent pattern set F, and the minimum support threshold *min_support*.

Output: The complete set of structured patterns.

Method: Call NaiveGraph$(\langle\rangle, 0, F)$.

Subroutine NaiveGraph(g, l, F)

The parameters are (1) g is a structured pattern; (2) l is the number of edges of g; and (3) F is the set of frequent patterns discovered so far.

Method:

1. Check whether g already existed in F, if yes, return; else insert g into F.
2. Scan S once, find every edge e such that e can be assembled to g to form a structured pattern $g \diamond_x e$.
3. For each structured pattern, $g \diamond_x e$, if it is frequent, call NaiveGraph$(g \diamond_x e, l+1, F)$.

NaiveGraph strictly follows the pattern-growth methodology: a pattern is grown by exploring locally frequent fragments. NaiveGraph shares the same processing steps with PrefixSpan (database projection is omitted). NaiveGraph is simple, but not efficient. The key issue is the inefficiency of extending g to $g \diamond_x e$. The same graph can be extended in different ways. For an n-edge graph, it may have n different ways to be built from $(n-1)$-edge graphs if we do not consider isomorphism. We call a graph that is discovered again a *duplicate graph*. The first step in Algorithm 2 gets rid of duplicate graphs. The number of duplicate graphs may be huge. It raises some severe problems. First, the generation and support computation of duplicate graphs waste time. Second, it is nontrivial to tell whether a graph is a duplicate. Generally, we have to compute its canonical label and check whether it was discovered before. Third, should we extend g if we find g is a duplicate? If there exists at least one graph that can grow only from this duplicate graph, we still need to extend it. As we can see, these three interleaved problems affect the efficiency of the algorithm. They suggest that g should be extended as conservatively as possible in order to reduce the generation of duplicate graphs. To satisfy this requirement, we developed gSpan where an efficient canonical labeling system and a lexicographic ordering in graphs are built. gSpan has the following salient properties: (1) it reduces the generation of duplicate graphs; (2) it does not need to search previous discovered frequent graphs in order to detect duplicates; and (3) it never extends any duplicate graph but still guarantees the completeness.

In the following sections, we introduce several techniques developed to represent and extend graphs efficiently. It includes mapping a graph to a DFS code (a sequence), building a lexicographic ordering among these codes, and mining DFS codes based on this lexicographic order.

4.1 DFS Subscripting

One can build a DFS tree when performing a

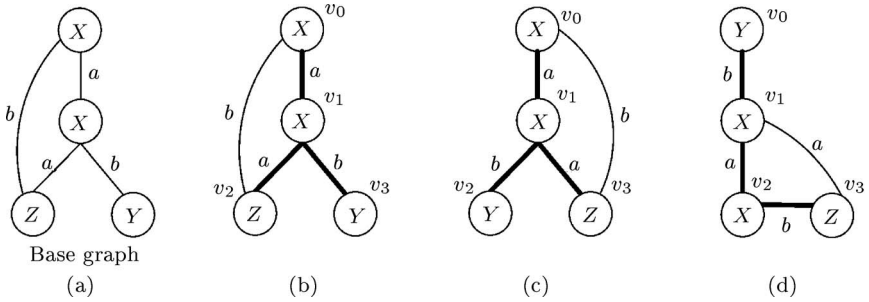

Fig.4. DFS subscripting.

depth-first search[21] in a graph. Fig.4(a) is a frequent subgraph for the sample dataset S in Fig.3. Figs.4(a)–4(d) are the same graph. The darkened edges in Figs.4(b)–4(d) compose three different DFS trees for this graph. When building a DFS tree, the depth-first discovery of the vertices forms a linear order. We use the magnitude of subscripts to illustrate this order according to their discovery time[21]. $i < j$ means v_i is discovered before v_j. We denote G subscripted with a DFS tree T by G_T. T is named a *DFS subscripting* of G.

Given G_T, we call the first node traversed in G_T, v_0, the *root*, and the last node traversed, v_n, the *right-most vertex*. The straight path from v_0 to v_n is named the *right-most path*. In Figs.4(b)–4(d), three different subscriptings are generated. The right-most path is (v_0, v_1, v_3) in Figs.4(b) and 4(c), and (v_0, v_1, v_2, v_3) in Fig.4(d).

Given G_T, the *forward edge* (*tree edge*[21]) set contains all the edges in the DFS tree, denoted by E_T^f, and the *backward edge* (*back edge*[21]) set contains all the edges which are not in the DFS tree, denoted by E_T^b. For example, the darkened edges in Figs.4(b)–4(d) are forward edges while the undarkened ones are backward edges. From now on, (v_i, v_j) (simply written as (i, j)) is viewed as an ordered pair to represent an edge. If $(v_i, v_j) \in E(G)$ and $i < j$, it is a forward edge; otherwise, a backward edge. The *forward edge of* v_i means there exists a forward edge (i, j) and $i < j$. The *backward edge of* v_i means there exists a backward edge (i, j) and $i > j$. In Fig.4(b), $(1, 3)$ is the forward edge of v_1, but not of v_3, and $(2, 0)$ is the backward edge of v_2.

4.2 Right-Most Extension

In Algorithm 2, NaiveGraph requires extending g in any possible position, which will result in a huge number of duplicate graphs. We would like to show that there is a more clever way to extend graphs. gSpan restricts the extension as follows: Given g and a DFS tree T in g, e can be extended from the right-most vertex connecting to any other vertices on the right-most path (*backward extension*); or e can be extended from vertices on the right-most path and introduces a new vertex (*forward extension*). We call these two kinds of restricted extension as *right-most extension*, denoted by $g \diamond_r e$ (for simplicity, we omit T here). This restricted extension is different from $g \diamond_x e$ described in NaiveGraph.

Example 9. If we want to extend the graph in Fig.4(b), the backward extension candidates can be (v_3, v_0). The forward extension candidates can be edges extending from v_3, v_1, or v_0 with a new vertex introduced.

Since we may have different DFS subscriptings for the same graph, we want to select one from them as *base subscripting* and conduct right-most extension on the base subscripting. Otherwise, right-most extension cannot reduce the generation of duplicate graphs because there are many different subscriptings to extend from the same graph.

4.3 DFS Code

For each subscripted graph, we can map it into an edge sequence. We build an order among these sequences and select the subscripting that generates the minimum sequence as its base subscripting. There are two kinds of orders in this process: (1) edge order, which maps edges in a subscripted graph into a sequence; and (2) sequence order, which builds the order among sequences. We introduce edge order in this subsection and sequence order in the next subsection.

Intuitively, the DFS tree has defined the discovery order of forward edges. For the graph shown in Fig.4(b), the forward edges are discovered in the order $(0, 1), (1, 2), (1, 3)$. Now we can insert backward edges into the order. Given a vertex v, all of its backward edges should appear just before its forward edges. If v does not have any forward edge, we put its backward edges just after the forward

edge where v is the second vertex. For vertex v_2 inFig.4(b), its backward edge $(2,0)$ should appear just after (1,2) since v_2 does not have any forward edge. Among the backward edges from the same vertex, we can enforce an order: Given v_i and its two backward edges, $(i,j_1),(i,j_2)$, if $j_1 < j_2$, then edge (i,j_1) will appear before edge (i,j_2). So far, we complete the ordering of the edges in a graph. Based on this order, we can translate a graph into a sequence. A complete sequence for Fig.4(b) is $(0,1),(1,2),(2,0),(1,3)$.

Formally we can define a linear order, $\prec_T$, in R^2 if we only consider the subscripts of edges. $e_1 \prec_T e_2$ holds if one of the following statements is true (assume $e_1 = (i_1, j_1), e_2 = (i_2, j_2)$):

(i) $e_1, e_2 \in E_T^f$, and $j_1 < j_2$ or $i_1 > i_2 \wedge j_1 = j_2$.

(ii) $e_1, e_2 \in E_T^b$, and $i_1 < i_2$ or $i_1 = i_2 \wedge j_1 < j_2$.

(iii) $e_1 \in E_T^b$, $e_2 \in E_T^f$, and $i_1 < j_2$.

(iv) $e_1 \in E_T^f$, $e_2 \in E_T^b$, and $j_1 \leqslant i_2$.

Example 10. For simplicity, we represent an edge by a 5-tuple, $(i, j, l_i, l_{(i,j)}, l_j)$, where l_i and l_j are the labels of v_i and v_j respectively and $l_{(i,j)}$ is the label of the edge between them. For example, (v_0, v_1) in Fig.4(b) is represented by $(0, 1, X, a, X)$. Table 4 shows the edge order for the DFS subscriptings in Figs.4(b)–4(d).

Table 4. DFS Code for Figs.4(b)–4(d)

edge	γ_0	γ_1	γ_2
e_0	$(0,1,X,a,X)$	$(0,1,X,a,X)$	$(0,1,Y,b,X)$
e_1	$(1,2,X,a,Z)$	$(1,2,X,b,Y)$	$(1,2,X,a,X)$
e_2	$(2,0,Z,b,X)$	$(1,3,X,a,Z)$	$(2,3,X,b,Z)$
e_3	$(1,3,X,b,Y)$	$(3,0,Z,b,X)$	$(3,1,Z,a,X)$

Definition 2 (DFS Code). *Given G_T, an edge sequence (e_i) can be constructed based on $\prec_T$, such that $e_i \prec_T e_{i+1}$, where $i = 0, \ldots, |E| - 1$. (e_i) is a DFS code, denoted by code(G, T).*

Table 4 shows three different DFS codes generated by DFS subscriptings in Figs.4(b)–4(d). As one can see, actually we build a one-to-one mapping between a subscripted graph and a DFS code. When the context is clear, we treat a subscripted graph and its DFS code as the same. All the notations on subscripted graphs can also be applied to DFS codes. The graph represented by a DFS code α is written as g_α.

4.4 DFS Lexicographic Order

We want to build an order among the DFS codes generated for a graph so that we can define a minimum DFS code for this graph. Since we are dealing with labeled graphs, the label information should be considered as one of the ordering factors, which can be used to break a tie when two edges have the same subscript, but different labels. We let $\prec_T$ take the first priority, the vertex label l_i take the second priority, the edge label $l_{(i,j)}$ take the third, and the vertex label l_j take the fourth to determine the order of two edges. For example, the first edges for the three DFS codes shown in Table 4 are $(0,1,X,a,X)$, $(0,1,X,a,X)$, and $(0,1,Y,b,X)$ respectively. All of them share the same $(0,1)$ subscript. So $\prec_T$ cannot tell the difference among them. But using label information, following the order of first vertex label, edge label, and second vertex label, we have $(0,1,X,a,X) < (0,1,Y,b,X)$. Based on this order, given DFS codes $\alpha = (a_0, a_1, \ldots, a_m)$ and $\beta = (b_0, b_1, \ldots, b_n)$, if $a_0 = b_0, \ldots, a_{t-1} = b_{t-1}$ and $a_t < b_t$ $(t \leqslant \min(m,n))$, then we say $\alpha < \beta$. According to this order definition, we have $\gamma_0 < \gamma_1 < \gamma_2$ for the DFS codes listed in Table 4.

The above discussion builds an order on the DFS codes of the same graph. We can extend this order definition in the DFS codes of different graphs. This ordering is one of our key contributions in gSpan. The formal definition of DFS code order is given as follows.

Definition 3 (DFS Lexicographic Order). *Suppose $Z = \{code(G,T) | T$ is a DFS subscripting of $G\}$, i.e., Z is a set containing all DFS codes of all connected labeled graphs. Suppose there is a linear order $(\prec_L)$ in the label set (L), then the lexicographic combination of $\prec_T$ and $\prec_L$ is a linear order $(<)$ in $N^2 \times L \times L \times L$ (the space of $(i, j, l_i, l_{(i,j)}, l_j)$).* **DFS Lexicographic Order** *is a linear order defined as follows. If $\alpha = code(G_\alpha, T_\alpha) = (a_0, a_1, \ldots, a_m)$ and $\beta = code(G_\beta, T_\beta) = (b_0, b_1, \ldots, b_n)$, $\alpha, \beta \in Z$, then $\alpha \leqslant \beta$ iff either of the following is true.*

(i) $\exists t$, $0 \leqslant t \leqslant \min(m,n)$, $a_k = b_k$ *for* $k < t, a_t < b_t$;

(ii) $a_k = b_k$ *for* $0 \leqslant k \leqslant m$, *and* $m \leqslant n$.

Example 11. Assume we have a 2-edge graph which has a DFS code $((0,1,X,a,X)\ (1,2,X,b,X))$. This graph is different from the graph in Fig.4(a). Using the DFS lexicographic order, we can compare $((0,1,X,a,X)\ (1,2,X,b,X))$ with any code in Table 4. It is greater than γ_0, but smaller than γ_1.

Definition 4 (Minimum DFS Code). *Given G, $Z(G) = \{code(G,T) | T$ is a DFS subscripting of $G\}$. Based on DFS lexicographic order, $\min(Z(G))$ is called* **Minimum DFS Code** *of G.*

If code$(G, T_0) = \min(Z(G))$, *we call* T_0 *the base subscripting of* G.

Code γ_0 in Table 4 is the minimum DFS code of the graph in Fig.4(a). We use $\min(\alpha)$ to denote the minimum DFS code of the graph represented by code α. Minimum DFS code can be considered as canonical label.

So far, we defined DFS code, minimum DFS code, and base subscripting. For every graph, we only conduct the right-most extension on its base subscripting and ignore other possible subscriptings. From now on, *the right-most extension of* G *specifically means the right-most extension on the base subscripting of* G. Can this extension method guarantee the completeness of the mining result? The answer is "*yes*". We first have the following result.

Theorem 1 (Completeness). *Performing right-most extension in* NaiveGraph *guarantees the completeness of mining result.*

When performing the right-most extension in NaiveGraph, it is possible that α is the minimum (i.e., representing a base subscripting), but $\alpha \diamond_r e$ is not. In this case, should we conduct the right-most extension on this non-minimum DFS code (i.e., it is not a base subscripting)? The answer is "*no*".

Lemma 2. *Performing only the right-most extension on the minimum DFS codes in* NaiveGraph *guarantees the completeness of the mining result.*

We achieved these two major results in gSpan. The detailed proof and implementation are available in [22]. Algorithm 3 outlines the framework. The difference between gSpan and NaiveGraph is the right-most extension and the termination condition on non-minimum DFS codes (Algorithm 3 lines 1–2). We replace the existence judgement in Step 1, Algorithm 2 with the inequation $s \neq \min(s)$. Actually, $s \neq \min(s)$ is more efficient to calculate.

Fig.5 shows the search space of gSpan, where each link represents a possible right-most extension. The right-most extension takes place when we extend $(k-1)$-edge graphs to the k-edge ones. If we find two DFS codes s and s' represent the same graph and $s < s'$, by Lemma 2, we can completely stop searching any descendant of s'. gSpan can generate graphs strictly in DFS lexicographic order: graphs with smaller minimum DFS codes will be discovered first.

Algorithm 3 (gSpan). Graph-based structured pattern mining.

Input: A graph database S, the frequent pattern set F, and the minimum support threshold *min_support*.

Output: The complete set of structured patterns.

Method: Call gSpan$(\langle\rangle, 0, F)$.

Subroutine gSpan(s, l, F)

The parameters are (1) s is a DFS code (a structured pattern); (2) l is the number of edges of s; and (3) F is the set of frequent patterns discovered so far.

Method:

1. Check whether s is equal to $\min(s)$, if yes, return; else insert s into F.
2. Scan S once, find every edge e such that s can be *right-most* extended with edge e to form a structured pattern $g \diamond_r e$.
3. For each structured pattern, $g \diamond_r e$, if it is frequent, call gSpan$(g \diamond_r e, l+1, F)$.

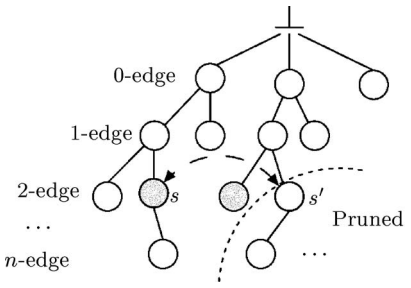

Fig.5. Search space.

5 Experimental Results and Performance Analysis

Since GSP[2] and SPADE[13] 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 gSpan.

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 512MB 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[13].

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

For real data set, we obtained the *Gazelle* data set from Blue Martini Software. This data set is used in KDD-CUP'2000 and contains totally 29,369 customers' webpage click-stream data. 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 29,369 sequences (i.e., customers), 35,722 sessions (i.e., transactions or events), and 87,546 page views (i.e., products or items). There are in total 1,423 distinct page views. More detailed information about this data set can be found in [23].

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[1] 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 $10,000$. 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 [13]. Additional data sets are used for scalability study and for testing the algorithm behavior with varied (and sometimes very low) support thresholds.

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 $1,000$. 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. Fig.6 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. Fig.7 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 = 6.8s) is about two orders of magnitude faster than GSP (runtime = 772.72s). When *min_support* is reduced to 0.5%, the data set contains a large number of frequent sequences, PrefixSpan takes 32.56s, which is more than 3.5 times faster than SPADE (116.35s), while GSP never terminates on our machine.

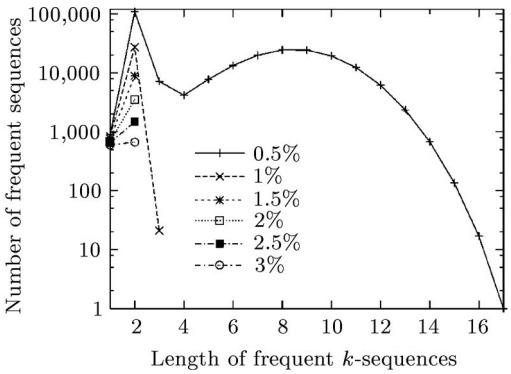

Fig.6. Distribution of frequent sequences of data set $C10T8S8I8$.

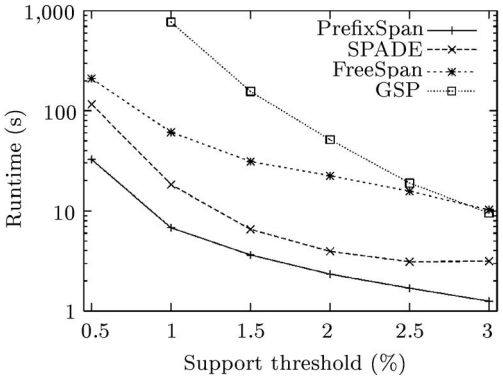

Fig.7. Performance of the four algorithms on data set $C10T8S8I8$.

The performance study on the real data set *Gazelle* is reported as follows. Fig.8 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. Fig.9 shows the per-

formance comparison among the four algorithms for Gazelle dataset. From Fig.9 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.

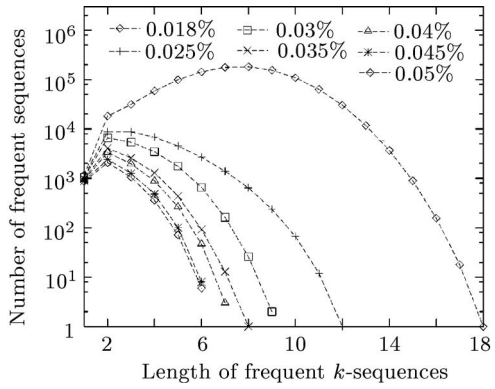

Fig.8. Distribution of frequent sequences of data set Gazelle.

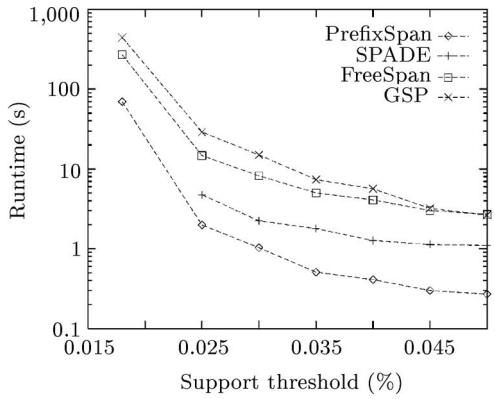

Fig.9. Performance of the four algorithms on data set Gazelle.

Finally, we compare the memory usage among the three algorithms, PrefixSpan, SPADE, and GSP using both real data set Gazelle and synthetic data set $C200T5S10I2.5$. Fig.10 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 40MB memory and SPADE just cannot stop running after it has used more than 22MB memory while PrefixSpan only uses about 2.7MB memory.

Fig.11 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 362MB 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 262MB memory, while PrefixSpan only uses 108MB memory. This also explains why in several cases in our previous experiments when the support threshold becomes really low, only PrefixSpan can finish running.

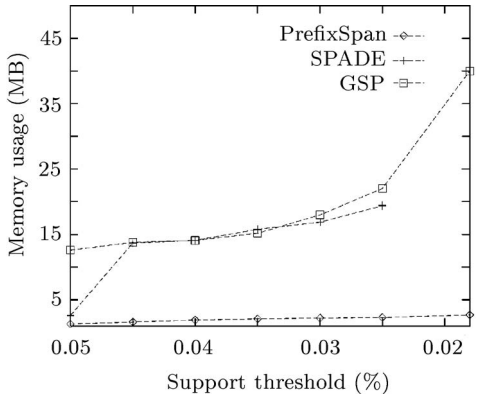

Fig.10. Memory usage comparison among PrefixSpan, SPADE, and GSP for data set Gazelle.

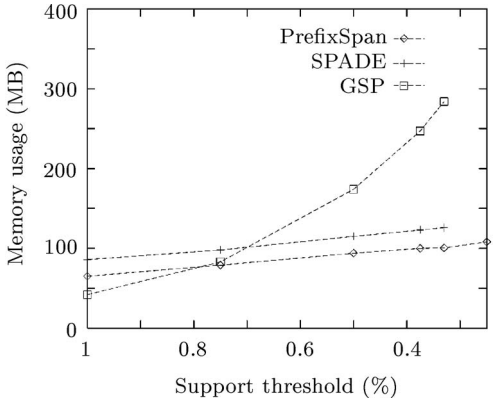

Fig.11. Memory usage: PrefixSpan, SPADE, and GSP for synthetic data set $C200T5S10I2.5$.

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 65MB to 108MB 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.

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 [13]. GSP performs fairly well only when *min_support* is rather high, with good scalability, which is consistent with the performance study reported in [2]. 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 become really slow. It is 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.

5.2 Performance Study of gSpan

We compared the performance of gSpan with FSG, an Apriori-based structured pattern mining algorithm. All the experiments are done on a 1.7GHz Intel Pentium-4 PC with 1GB main memory, running RedHat 7.3. gSpan is implemented in C++ with STL library support and compiled by g++ with -O3 optimization. In each experiment, we also show the performance of FSG, which is kindly provided by Kuramochi *et al.*

The data set we tested is an AIDS antiviral screen chemical compound dataset[②], which comes from Developmental Theroapeutics Program in NCI/NIH. We select the most up-to-date release, March 2002 Release. The dataset contains 43,905 chemical compounds. The results of the screening tests can be categorized into three classes: **CA**: confirmed active; **CM**: confirmed moderately active; and **CI**: confirmed inactive. Among these 43,905 compounds, 423 of them belong to **CA**, 1,083 are of **CM**, and the remaining is in class **CI**.

We are interested in the frequent structures in class **CA** and **CM** compounds. All the hydrogens in these compounds are removed. The most popular atoms in these two datasets are C, O, N, S, etc. There are 21 kinds of atoms in class **CA** compounds whereas 25 in class **CM**. Three kinds of bonds are popular in these compounds: single-bond, double-bond, and aromatic-bond. On average, each class **CA** compound has 40 vertices and 42 edges. The maximum one has 188 vertices and 196 edges. Each class **CM** compound has 32 vertices and 34 edges on average. The maximum one has 221 vertices and 234 edges.

Fig.12(a) shows the runtime with *min_support* varying from 10% to 5%. As we can see, gSpan outperforms FSG by a factor of 6 when *min_support* is close to 5%. Fig.12(b) shows the memory consumption of these two algorithms. gSpan consumes much less main memory than FSG. The reduction is around 2 orders of magnitude. The largest pattern discovered with 5% support has 42 edges.

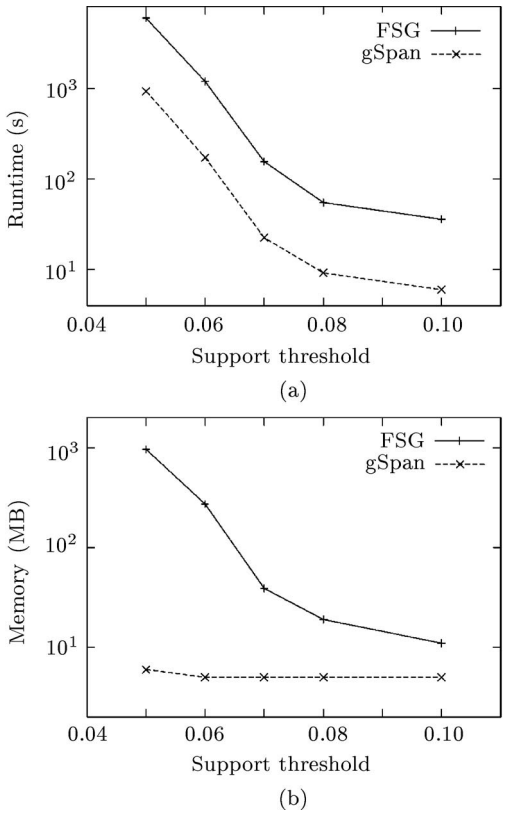

Fig.12. Mining patterns in class **CA** compounds. (a) Runtime. (b) Memory.

Next, we conduct experiments on class **CM** compounds. The performance is shown in Fig.13. The largest pattern with 5% support has 23 edges.

② http://dtp.nci.nih.gov/docs/aids/aids_data.html.

That means the compounds in class **CA** share larger chemical fragments. The compounds in class **CM** are more diverse.

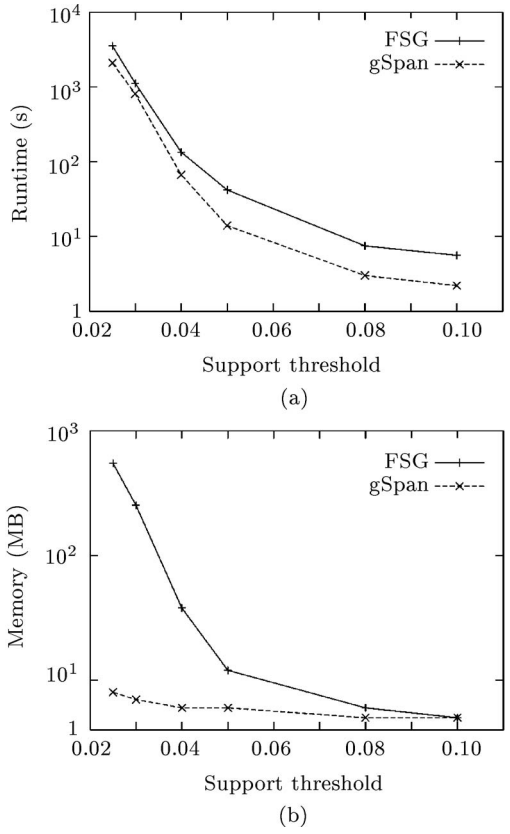

Fig.13. Mining patterns in class **CM** compounds. (a) Runtime. (b) Memory.

We also run experiments of gSpan and FSG on a series of synthetic datasets. The results are similar to the ones shown in Figs.12 and 13.

6 Extensions of Sequential and Structured Pattern Growth Approach

Compared with mining unordered frequent patterns, mining sequential and structured patterns is an important step towards mining sophisticated frequent patterns in large databases. With the successful development of sequential pattern-growth and structured pattern-growth methods, 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 pattern-growth approach.

6.1 Mining Multi-Dimensional, Multi-Level Sequential and Structured 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 she/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, [1, 24], and mining association or correlations at multiple dimensional space, [25, 26]. 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 12 (Mining multi-dimensional, multi-level sequential patterns). Consider a sequence database *SDB* in Table 5, 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*.

Table 5. A Multi-Dimensional Sequence Database

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	$\langle(be)(ce)\rangle$

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 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_X*30), (*Dell, PC, Precision_*330)) (*Canon, digital_camera, CD*420), (*IBM, Laptop, Thinkpad_X*30), (*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 12, 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[28] and H-cubing[29], 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.

Similar applications also exist in structured pattern mining. For example, chemical compounds may change their activities according to temperature, pressure, density, and so on. At coarse levels, some structures may share the similar behavior. But at fine levels, they may behave differently. Moreover, different structure patterns can be associated with multi-dimensional features, such as different categories, different time durations, etc. Thus it is important to mine structured patterns in the multi-level, multi-dimensional context. Although the methodology of extension of single-level, single-dimensional mining algorithm towards multi-level, multi-dimensional mining could be similar to sequential and structured patterns. It is still a research issue to develop efficient and scalable algorithms for structured patterns.

6.2 Constraint-Based Mining of Sequential and Structured 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[30−32]. 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[32]. 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[12] can be handled elegantly. Let us examine one such example.

Example 13 (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).

Since a regular expression constraint, like C, is neither anti-monotone, nor monotone, nor succinct, the classical constraint-pushing framework[30] cannot push it deep. To overcome this difficulty, Garofalakis, *et al.*[12] 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[2], 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 sequen-

tial pattern mining process[33]. 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 13 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 [33].

In structured pattern mining, we may apply the wildcard constraint in gSpan using the similar integration method introduced above. gSpan is designed for connected undirected structured pattern mining. There are other kinds of interesting structured patterns. With little modification, gSpan can be extended to mine those patterns. For example, mining non-simple graphs, non-simple graphs may contain self-loops (i.e., an edge joining a vertex to itself) and multiple edges (i.e., several edges connecting to the same two vertices). In gSpan, we always first grow backward edges and then forward edges. In order to accommodate self-loops, the growing order can be changed to *backward edges, self-loops, and forward edges.* Multiple edges can appear in these three kinds of edges. If we allow two neighboring edges in a DFS code to share the same vertices, the definition of DFS lexicographic order can be extended to include multiple edges. Thus gSpan can mine non-simple graphs efficiently as well. By removing backward edges, gSpan is ready to mine tree structures. The efficiency of this dwarfed version of frequent graph mining was demonstrated in [34].

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[2] and SPADE[13], we promote a divide-and-conquer approach, called *pattern-growth approach*, which is an extension of FP-growth[20], 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. 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.

By employing the similar pattern-growth methodology, gSpan can mine structured patterns efficiently. The DFS coding technique developed in gSpan can smoothly transform a structured pattern to a sequential pattern. With this coding technique, we can reuse the mining framework of PrefixSpan in structured pattern mining and demonstrate its efficiency again.

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.

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

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

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

[4] Wang J, Chirn G, Marr T, Shapiro B, Shasha D, Zhang K. Combinatorial pattern discovery for scientific data: Some preliminary results. In *Proc. 1994 ACM-SIGMOD Int. Conf. Management of Data* (*SIGMOD'94*), Minneapolis, MN, May, 1994, pp.115–125.

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

[6] Zaki M J. Efficient enumeration of frequent sequences. In *Proc. 7th Int. Conf. Information and Knowledge Management* (*CIKM'98*), Washington D.C., Nov. 1998, pp.68–75.

[7] Masseglia F, Cathala F, Poncelet P. The psp approach for mining sequential patterns. In *Proc. 1998 European Symp. Principle of Data Mining and Knowledge Discovery* (*PKDD'98*), Nantes, France, Sept. 1998, pp.176–184.

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

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

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

[11] Ramaswamy S, Mahajan S, Silberschatz A. On the discovery of interesting patterns in association rules. In *Proc. 1998 Int. Conf. Very Large Data Bases* (*VLDB'98*), New York, NY, Aug. 1998, pp.368–379.

[12] Guha S, Rastogi R, Shim K. Rock: A robust clustering algorithm for categorical attributes. In *Proc. 1999 Int. Conf. Data Engineering* (*ICDE'99*), Sydney, Australia, Mar. 1999, pp.512–521.

[13] Zaki M. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning*, 2001, 40: 31–60.

[14] Zaki M J, Hsiao C J. CHARM: An efficient algorithm for closed itemset mining. In *Proc. 2002 SIAM Int. Conf. Data Mining* (*SDM'02*), Arlington, VA, April 2002, pp.457–473.

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

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

[17] Inokuchi A, Washio T, Motoda H. An apriori-based algorithm for mining frequent substructures from graph data. In *Proc. 2000 European Symp. Principle of Data Mining and Knowledge Discovery* (*PKDD'00*), Lyon, France, Sept. 1998, pp.13–23.

[18] Kuramochi M, Karypis G. Frequent subgraph discovery. In *Proc. 2001 Int. Conf. Data Mining* (*ICDM'01*), San Jose, CA, Nov. 2001, pp.313–320.

[19] Vanetik N, Gudes E, Shimony S E. Computing frequent graph patterns from semistructured data. In *Proc. 2002 Int. Conf. Data Mining* (*ICDM'02*), Maebashi, Japan, Dec. 2002, pp.458–465.

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

[21] Cormen T, Leiserson C, Rivest R, Stein C. Introduction to Algorithms, 2nd ed. The MIT Press, Cambridge, MA, 2001.

[22] Yan X, Han J. gSpan: Graph-based substructure pattern mining. In *UIUC-CS Tech. Report: R-2002-2296*, A 4-page short version published in *Proc. 2002 Int. Conf. Data Mining* (*ICDM'02*), Maebashi, Japan, 2002, pp.721–724.

[23] Kohavi R, Brodley C, Frasca B, Mason L, Zheng Z. KDD-Cup 2000 organizers' report: Peeling the onion. *SIGKDD Explorations*, 2000, 2: 86–98.

[24] Han J, Fu Y. Discovery of multiple-level association rules from large databases. In *Proc. 1995 Int. Conf. Very Large Data Bases* (*VLDB'95*), Zurich, Switzerland, Sept. 1995, pp.420–431.

[25] Kamber M, Han J, Chiang J Y. Metarule-guided mining of multi-dimensional association rules using data cubes. In *Proc. 1997 Int. Conf. Knowledge Discovery and Data Mining* (*KDD'97*), Newport Beach, CA, Aug. 1997, pp.207–210.

[26] Grahne G, Lakshmanan L V S, Wang X, Xie M H. On dual mining: From patterns to circumstances, and back.

In *Proc. 2001 Int. Conf. Data Engineering* (*ICDE'01*), Heidelberg, Germany, April 2001, pp.195–204.

[27] Pinto H, Han J, Pei J, Wang K, Chen Q, Dayal U. Multi-dimensional sequential pattern mining. In *Proc. 2001 Int. Conf. Information and Knowledge Management* (*CIKM'01*), Atlanta, GA, Nov. 2001, pp.81–88.

[28] Beyer K, Ramakrishnan R. Bottom-up computation of sparse and iceberg cubes. In *Proc. 1999 ACM-SIGMOD Int. Conf. Management of Data* (*SIGMOD'99*), Philadelphia, PA, June 1999, pp.359–370,

[29] Han J, Pei J, Dong G, Wang K. Efficient computation of iceberg cubes with complex measures. In *Proc. 2001 ACM-SIGMOD Int. Conf. Management of Data* (*SIGMOD'01*), Santa Barbara, CA, May 2001, pp.1–12.

[30] Ng R, Lakshmanan L V S, Han J, Pang A. Exploratory mining and pruning optimizations of constrained associations rules. In *Proc. 1998 ACM-SIGMOD Int. Conf. Management of Data* (*SIGMOD'98*), Seattle, WA, June 1998, pp.13–24.

[31] Bayardo R J, Agrawal R, Gunopulos D. Constraint-based rule mining on large, dense data sets. In *Proc. 1999 Int. Conf. Data Engineering* (*ICDE'99*), Sydney, Australia, April 1999, 188–197.

[32] Pei J, Han J, Lakshmanan L V S. Mining frequent itemsets with convertible constraints. In *Proc. 2001 Int. Conf. Data Engineering* (*ICDE'01*), Heidelberg, Germany, April 2001, pp.433–442.

[33] Pei J, Han J, Wang W. Constraint-based sequential pattern mining in large databases. In *Proc. 2002 Int. Conf. Information and Knowledge Management* (*CIKM'02*), McLean, VA, Nov. 2002, pp.18–25.

[34] Asai T, Abe K, Kawasoe S, Arimura H, Satamoto H, Arikawa S. Efficient substructure discovery from large semi-structured data. In *Proc. 2002 SIAM Int. Conf. Data Mining* (*SDM'02*), Part III, Arlington, VA, April 2002.

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

[36] Yan X, Han J. gSpan: Graph-based substructure pattern mining. In *Proc. 2002 Int. Conf. Data Mining* (*ICDM'02*), Maebashi, Japan, Dec. 2002, pp.721–724.

[37] Yan X, Han J. 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.

Jia-Wei Han is a professor in computer science, University of Illinois at Urbana-Champaign. He has been working on research into data mining, data warehousing, database systems, with over 250 conference and journal publications. He has chaired or served on the PCs in many international conferences, including ACM SIGKDD, ACM SIGMOD, VLDB, ICDE, ICDM, SDM, and EDBT. He also served or is serving on the editorial boards for Data Mining and Knowledge Discovery, IEEE Transactions on Knowledge and Data Engineering, Journal of Intelligent Information Systems, and Journal of Computer Science and Technology. He is currently serving on the Board of Directors for the Executive Committee of ACM Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD). Jiawei has received the Outstanding Contribution Award at the 2002 ICDM, ACM Service Award, and IBM Faculty Awards. He is an ACM Fellow and the first author of the textbook "Data Mining: Concepts and Techniques" (Morgan Kaufmann, 2001).

Jian Pei received the B. Eng. and the M. Eng. degrees, both in computer science, from Shanghai Jiaotong University, China, in 1991 and 1993, respectively, and the Ph.D. degree in computing science from Simon Fraser University, Canada, in 2002. He was a Ph.D. candidate in Peking University in 1997–1999.

He is currently an Assistant professor of computer Science and engineering, the State University of New York at Buffalo, USA. He is a participating faculty in the Center of Unified Biometrics and Sensors (CUBS), at State University of New York at Buffalo. His research interests include data mining, data warehousing, online analytical processing, database systems, and bioinformatics. His current research is supported in part by the National Science Foundation (NSF).

He has published over 40 research papers in refereed journals, conferences, and workshops, served in the program committees of over 30 international conferences and workshops, and been a reviewer for some leading academic journals. He is a member of the ACM, the ACM SIGMOD, the ACM SIGKDD and the IEEE Computer Society, and a guest area editor of the Journal of Computer Science and Technology.

Xi-Feng Yan received a B.E. degree in computer engineering from Zhejiang University, China, in 1997, and an M.S. degree in computer science from the State University of New York at Stony Brook, NY, in 2001. He is currently a Ph.D. candidate in the Department of Computer Science at the University of Illinois at Urbana-Champaign. His research interests include data mining, structural/graph pattern mining, and their applications in database systems and bioinformatics.