Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach

Jian Pei, *Member*, *IEEE Computer Society*, Jiawei Han, *Senior Member*, *IEEE*, Behzad Mortazavi-Asl, Jianyong Wang, Helen Pinto, Qiming Chen, Umeshwar Dayal, *Member*, *IEEE Computer Society*, and Mei-Chun Hsu

Abstract—Sequential pattern mining is an important data mining problem with broad applications. However, it is also a difficult problem since the mining may have to generate or examine a combinatorially explosive number of intermediate subsequences. Most of the previously developed sequential pattern mining methods, such as *GSP*, explore a *candidate generation-and-test* approach [1] to reduce the number of candidates to be examined. However, this approach may not be efficient in mining large sequence databases having numerous patterns and/or long patterns. In this paper, we propose a projection-based, sequential pattern-growth approach for efficient mining of sequential patterns. In this approach, a sequence database is recursively projected into a set of smaller projected databases, and sequential patterns are grown in each projected database by exploring only locally frequent fragments. Based on an initial study of the pattern growth-based sequential pattern mining, *FreeSpan* [8], we propose a more efficient method, called PSP, which offers ordered growth and reduced projected databases. To further improve the performance, a *pseudoprojection* technique is developed in *PrefixSpan*. A comprehensive performance study shows that *PrefixSpan*, in most cases, outperforms the a priori-based algorithm *GSP*, *FreeSpan*, and *SPADE* [29] (a sequential pattern mining algorithm that adopts vertical data format), and *PrefixSpan* integrated with pseudoprojection is the fastest among all the tested algorithms. Furthermore, this mining methodology can be extended to mining sequential patterns with user-specified constraints. The high promise of the pattern-growth approach may lead to its further extension toward efficient mining of other kinds of frequent patterns, such as frequent substructures.

Index Terms—Data mining algorithm, sequential pattern, frequent pattern, transaction database, sequence database, 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, etc.

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

- *J. Pei is with the School of Computing Science, Simon Fraser University, Canada. E-mail: jpei@cse.sfu.edu.*
- *J. Han is with the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801. E-mail: hanj@cs.uiuc.edu.*
- *J. Wang is with the University of Minnesota, Minneapolis, MN 55455. E-mail: jianyong@cs.umn.edu.*
- *B. Mortazavi-Asl and H. Pinto are with the School of Computing Science, Simon Fraser University, Canada. E-mail: {mortazav, hlpinto}@cs.sfu.ca.*
- *Q. Chen is with PacketMotion Inc., San Mateo, CA 94403.*
- *E-mail: qchen@packetmotion.com.*
- *U. Dayal is with Hewlett-Packard Labs., 1501 Page Mill Road, Palo Alto, CA 94303. E-mail: dayal@hpl.hp.com.*
- *M.-C. Hsu is with Commerce One Inc., San Francisco, CA 94105. E-mail: meichun.hsu@commerceone.com.*

Manuscript received 6 Feb. 2003; revised 30 July 2003; accepted 9 Oct. 2003. For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 118248.

Many previous studies contributed to the efficient mining of sequential patterns or other frequent patterns in time-related data [2], [23], [13], [24], [30], [14], [12], [5], [16], [22], [7]. Srikant and Agrawal [23] generalized their definition of sequential patterns in [2] to include time constraints, sliding time window, and user-defined taxonomy, and presented an a priori-based, improved algorithm *GSP* (i.e., *generalized sequential patterns*). Mannila et al. [13] 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 intertransaction association rules. These are essentially rules whose left-hand and right-hand sides are episodes with time-interval restrictions. Lu et al. [12] proposed intertransaction association rules that are implication rules whose two sides are totally-ordered episodes with timing-interval restrictions. Guha et al. [6] 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. [16] 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 percent confidence. Han et al. [7] developed a frequent pattern mining method for mining partial periodicity patterns that

are frequent maximal patterns where each pattern appears in a fixed period with a fixed set of offsets and with sufficient support.

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

A typical a priori-like sequential pattern mining method, such as *GSP* [23], adopts a multiple-pass, candidate generation-and-test approach outlined as follows: The first scan finds all of the frequent items that 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*, based on the a priori principle. Each candidate sequence contains one more item than a seed sequential pattern, where each element in the pattern may contain one item 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 the candidates with support no less than min_support in the database form the set of the newly found sequential patterns. This set is then used as the seed set for the next pass. The algorithm terminates when no new sequential pattern is found in a pass, or when no candidate sequence can be generated.

The a priori-like sequential pattern mining method, though reducing search space, bears three nontrivial, inherent costs that are independent of detailed implementation techniques.

- **A huge set of candidate sequences could be generated in a large sequence database**. Since the set of candidate sequences includes all the possible permutations of the elements and repetition of items in a sequence, the a priori-based method may generate a really large set of candidate sequences even for a moderate seed set. For example, two frequent sequences of length-1, $\langle a \rangle$ and $\langle b \rangle$, will generate five candidate sequences of length-2: $\langle aa \rangle$, $\langle ab \rangle$, $\langle ba \rangle$, $\langle bb \rangle$, and $\langle (ab) \rangle$, where $\langle (ab) \rangle$ represents that two events, a and b, happen in the same time slot. If there are 1,000 frequent sequences of length-1, such as $\langle a_1 \rangle, \langle a_2 \rangle, \ldots, \langle a_{1000} \rangle$, an a priori-like algorithm will generate $1,000 \times 1,000 + \frac{1,000 \times 999}{2} = 1,499,500$ candidate sequences. Notice that the cost of candidate sequence generation, test, and support counting is inherent to the a priori-based method, no matter what technique is applied to optimize its detailed implementation.
- **Multiple scans of databases in mining.** The length of each candidate sequence grows by one at each database scan. In general, to find a sequential pattern of length l, the a priori-based method must scan the database at least l times. This bears a nontrivial cost when long patterns exist.
- **The a priori-based method generates a combinatorially explosive number of candidates when mining long sequential patterns.** A long sequential pattern contains a combinatorial explosive number of subsequences, and such subsequences must be generated and tested in the a priori-based mining. Thus, the number of candidate sequences is exponential to the length of the sequential patterns to be mined. For example, let the database contain only one single sequence of length 100, $\langle a_1 a_2 \ldots a_{100} \rangle$, and the *min_support* threshold be 1 (i.e., every occurring pattern is frequent). To (re)derive this length-100 sequential pattern, the a priori-based method has to generate 100 length-1 candidate sequences (i.e., $\langle a_1 \rangle$, $\langle a_2 \rangle, \ldots, \langle a_{100} \rangle$), $100 \times 100 + \frac{100 \times 99}{2} = 14,950$ length-2 candidate sequences, $\binom{100}{3} = 161,700$ length-3 candidate sequences,[1] and so on. Obviously, the total number of candidate sequences to be generated is $\Sigma_{i=1}^{100} \binom{100}{i} = 2^{100} - 1 \approx 10^{30}$.

In many applications, such as DNA analysis or stock sequence analysis, the databases often contain a large number of sequential patterns and many patterns are long. It is important to reexamine the sequential pattern mining problem to explore more efficient and scalable methods.

Based on our observation, both the thrust and the bottleneck of an a priori-based sequential pattern mining method come from its step-wise candidate sequence generation and test. Can we develop a method that may absorb the spirit of a priori, but avoid or substantially reduce the expensive candidate generation and test? This is the motivation of this study.

In this paper, we systematically explore 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 databases by exploring only locally frequent fragments*. Based on this philosophy, we first proposed a straightforward pattern growth method, *FreeSpan* (for **Fre**qu**e**nt pattern-projected **S**equential **pa**ttern mining) [8], which reduces the efforts of candidate subsequence generation. In this paper, we introduce 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 a priori-based algorithm *GSP*, *FreeSpan*, and *SPADE* [29] (a sequential pattern mining algorithm that adopts vertical data format) and *PrefixSpan*, integrated with pseudoprojection, is the fastest among all the tested algorithms. Furthermore, our experiments show that *PrefixSpan* consumes a much smaller memory space in

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

TABLE 1
A Sequence Database

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

comparison with *GSP* and *SPADE*. This pattern-growth methodology can be further extended to mining multilevel, multidimensional sequential patterns, and mining other structured patterns.

The remainder of the paper is organized as follows: In Section 2, the sequential pattern mining problem is defined, and the a priori-based sequential pattern mining method, *GSP*, is illustrated. In Section 3, our approach, projection-based sequential pattern growth, is introduced, by first summarizing *FreeSpan*, and then presenting *PrefixSpan*, associated with a pseudoprojection technique for performance improvement. Our experimental results and performance analysis are reported in Section 4. Some extensions of the method and future research issues are discussed in Section 5, and our study is concluded in Section 6.

2 Problem Definition and the *GSP* Algorithm

In this section, the problem of sequential pattern mining is defined, and the most representative a priori-based sequential pattern mining method, *GSP* [23], is illustrated using an example.

2.1 Problem Definition

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

A **sequence database** S is a set of tuples $\langle sid, s\rangle$, where sid is a **sequence_id** and s a sequence. A tuple $\langle sid, s\rangle$ is said to *contain* a sequence α, if α is a subsequence of s. The support of a sequence α in a sequence database S is the number of tuples in the database containing α, i.e.,

$$support_S(\alpha) = \mid \{\langle sid, s\rangle | (\langle sid, s\rangle \in S) \wedge (\alpha \sqsubseteq s)\} \mid .$$

It can be denoted as $support(\alpha)$ if the sequence database is clear from the context. Given a positive integer *min_support* as the **support threshold**, a sequence α is called a **sequential pattern** in sequence database S if $support_S(\alpha) \ge min\_support$. A sequential pattern with length l is called an ***l*-pattern**.

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

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 nine 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 1 to the *support* of $\langle a\rangle$. Also, sequence $\langle a(bc)df\rangle$ is a *subsequence* of $\langle a(abc)(ac)d(cf)\rangle$. Since both sequences 10 and 30 *contain* subsequence $s = \langle (ab)c\rangle$, s is a *sequential pattern* of length 3 (i.e., 3-*pattern*).

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

2.2 Algorithm *GSP*

As outlined in Section 1, a typical sequential pattern mining method, *GSP* [23], mines sequential patterns by adopting a candidate subsequence generation-and-test approach, based on the a priori principle. 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 multiscan 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 kth 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.

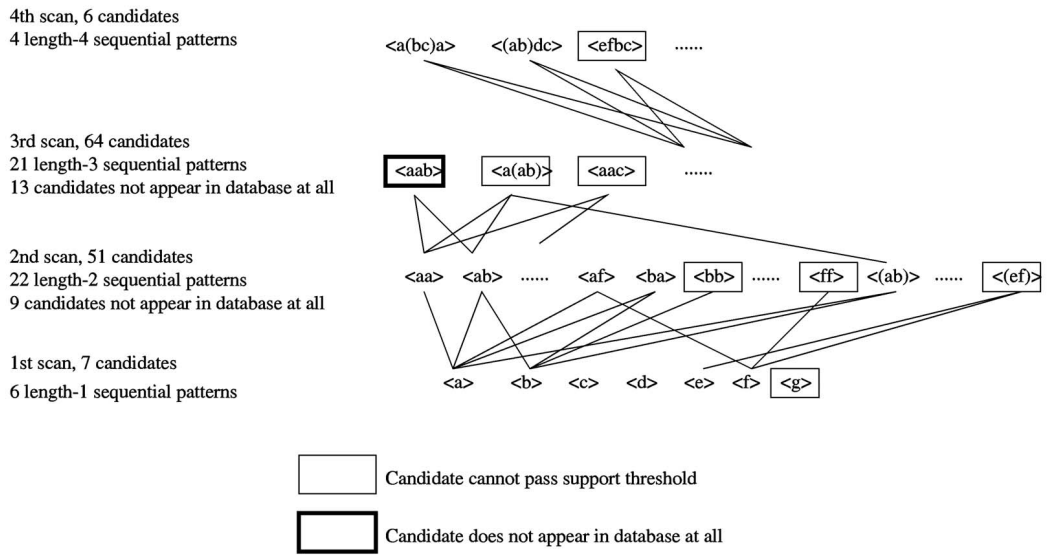


Fig. 1. Candidates and sequential patterns in *GSP*.

GSP, though benefits from the a priori 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.

3 Mining Sequential Patterns by Pattern Growth

As analyzed in Section 1 and Example 2, the *GSP* algorithm shares similar strengths and weaknesses as the a priori method. For frequent pattern mining, a frequent pattern growth method called *FP-growth* [9] has been developed for efficient mining of frequent patterns without candidate generation. The method uses a data structure called *FP-tree* to store compressed frequent patterns in transaction database and recursively mines the projected conditional *FP-trees* to achieve high performance.

Can we mine sequential patterns by extension of the FP-tree structure? Unfortunately, the answer cannot be so optimistic because it is easy to explore the sharing among a set of unordered items, but it is difficult to explore the sharing of common data structures among a set of ordered items. For example, a set of frequent itemsets $\{abc, cbad, ebadc, cadb\}$ share the same tree branch $\langle abcde\rangle$ in the *FP-tree*. However, if they were a set of sequences, there is no common prefix subtree structure that can be shared among them because one cannot change the order of items to form sharable prefix subsequences.

Nevertheless, one can explore the spirit of *FP-growth*: divide the sequential patterns to be mined based on the subsequences obtained so far and project the sequence database based on the partition of such patterns. Such a methodology is called *sequential pattern mining by pattern-growth*. The general idea is outlined as follows: *Instead of repeatedly scanning the entire database and generating and testing large sets of candidate sequences, one can recursively* **project** *a sequence database into a set of smaller databases associated with the set of patterns mined so far and, then, mine locally frequent patterns in each projected database.*

In this section, we first outline a projection-based sequential pattern mining method, called *FreeSpan* [8], and then systematically introduce an improved method *PrefixSpan* [19]. Both methods generate projected databases, but they differ at the criteria of database projection: *FreeSpan* creates projected databases based on the current set of frequent patterns without a particular ordering (i.e., growth direction), whereas *PrefixSpan* projects databases by growing frequent prefixes. Our study shows that, although both *FreeSpan* and *PrefixSpan* are efficient and scalable, *PrefixSpan* is substantially faster than *FreeSpan* in most sequence databases.

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 ith subset $(1 \le i \le n)$ 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 3 (*FreeSpan*). Given the database S and $min\_support$ in Example 1, *FreeSpan* first scans S, collects the support for each item, and finds the set of frequent items. This step is similar to *GSP*. Frequent items are listed in support descending order (in the form of "*item : support*"), that is, $f\_list = a:4,\ b:4,\ c:4,\ d:3,\ e:3,\ f:3$. They form six length-one sequential patterns:

$$\langle a\rangle:4, \langle b\rangle:4, \langle c\rangle:4, \langle d\rangle:3, \langle e\rangle:3, \langle f\rangle:3.$$

According to the f_list, the complete set of sequential patterns in S can be divided into six 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,
4. 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.** By mining the $\langle a\rangle$-*projected database*: $\{\langle aaa\rangle, \langle aa\rangle, \langle a\rangle, \langle a\rangle\}$, 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.** Mining 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 [8]. By the analysis of Example 3, we have the following observations on the strength and weakness of *FreeSpan*, which are also verified by our later experimental study.

On the one hand, *the strength of FreeSpan is that it searches a smaller projected database than GSP in each subsequent database projection.* This is because *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.

On the other hand, *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 of the same sequences as that in the original sequence database, except for the removal of the infrequent item g in sequence 40. 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. *Is it possible to reduce the size of projected database and the cost of checking at every possible position of a potential candidate sequence?* 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 10 in our running example is listed as $\langle a(abc)(ac)d(cf)\rangle$ instead of $\langle a(bac)(ca)d(fc)\rangle$. With such a convention, the expression of a sequence is unique.

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

Definition 1 (Prefix). *Suppose all the items within an element are listed alphabetically. Given a sequence $\alpha = \langle e_1 e_2 \cdots e_n\rangle$ (where each e_i corresponds to a frequent element in S), a*

sequence $\beta = \langle e'_1 e'_2 \cdots e'_m \rangle$ $(m \leq n)$ is called a prefix of α if and only if 1) $e'_i = e_i$ for $(i \leq m-1)$; 2) $e'_m \subseteq e_m$; and 3) all the frequent items in $(e_m - e'_m)$ are alphabetically after those in e'_m.

For example, $\langle a \rangle$, $\langle aa \rangle$, $\langle a(ab) \rangle$, 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.

Definition 2 (Suffix). *Given a sequence $\alpha = \langle e_1 e_2 \cdots e_n \rangle$ (where each e_i corresponds to a frequent element in S). Let $\beta = \langle e_1 e_2 \cdots e_{m-1} e'_m \rangle$ $(m \leq n)$ be the prefix of α. Sequence $\gamma = \langle e''_m e_{m+1} \cdots e_n \rangle$ is called the suffix of α with regards to prefix β, denoted as $\gamma = \alpha / \beta$, where $e''_m = (e_m - e'_m)$.[2] We also denote $\alpha = \beta \cdot \gamma$. Note, if β is not a subsequence of α, the suffix of α with regards to β is empty.*

For example, for the sequence $s = \langle a(abc)(ac)d(cf) \rangle$, $\langle (abc)(ac)d(cf) \rangle$ is the *suffix* with regards to the prefix $\langle a \rangle$, $\langle (\_bc)(ac)d(cf) \rangle$ is the *suffix* with regards to the prefix $\langle aa \rangle$, and $\langle (\_c)(ac)d(cf) \rangle$ is the *suffix* with regards to 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.

Lemma 3.1 (Problem partitioning).

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

Proof. We show the correctness of the second half of the lemma. The first half is a special case where $\alpha = \langle \rangle$.

For a sequential pattern γ with prefix α, where α is of length l, the length-$(l+1)$ prefix of γ must be a sequential pattern, according to the a priori heuristic. Furthermore, the length-$(l+1)$ prefix of γ is also with prefix α, according to the definition of prefix. Therefore, there exists some j $(1 \leq j \leq m)$ such that β_j is the length-$(l+1)$ prefix of γ. Thus, γ is in the jth subset. On the other hand, since the length-k prefix of a sequence γ is unique, γ belongs to only one determined subset. That is, the subsets are disjoint. So, we have the lemma. □

Based on Lemma 3.1, 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.

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

Definition 3 (Projected database). *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 with regards to prefix α.*

To collect counts in projected databases, we have the following definition:

Definition 4 (Support count in projected database). *Let α be a sequential pattern in sequence database S, and 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 3.2 (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 with regards to β, γ/β, equals to the sequence resulted from first doing projection of γ with regards to α, i.e., γ/α, and then doing projection γ/α with regards to β. That is $\gamma/\beta = (\gamma/\alpha)/\beta$.

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

The third part of the lemma is on the size of a projected database. Obviously, the α-projected database can have the same number of sequences as S only if α appears in every sequence in S. Otherwise, only those sequences in S which are super-sequences of α appear in the α-projected database. So, the α-projected database cannot contain more sequences than S. For every sequence γ in S such that γ is a super-sequence of α, γ appears in the α-projected database in whole only if α is a prefix of γ. Otherwise, only a subsequence of γ appears in the α-projected database. Therefore, the size of α-projected database cannot exceed that of S. □

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

Example 4 (*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:

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.

TABLE 2
Projected Databases and Sequential Patterns

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

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 3) the ones with prefix $\langle f \rangle$.
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.

      The sequences in S containing $\langle a \rangle$ are projected with regards to $\langle a \rangle$ 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 six subsets: 1) those prefixed with $\langle aa \rangle$, 2) those with $\langle ab \rangle$, $\ldots$, and, finally, 3) 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 nonempty (suffix) subsequences prefixed with $\langle aa \rangle$: $\{\langle (\_bc)(ac)d(cf) \rangle, \{\langle (\_e) \rangle\}$. Since there is no hope to generate any frequent subsequence from this projected database, the processing of the $\langle aa \rangle$-projected database terminates.
      ii. The $\langle ab \rangle$-projected database consists of three suffix sequences: $\langle (\_c)(ac)d(cf) \rangle$, $\langle (\_c)a \rangle$, and $\langle c \rangle$. Recursively mining the $\langle ab \rangle$-projected database returns four sequential patterns: $\langle (\_c) \rangle$, $\langle (\_c)a \rangle$, $\langle a \rangle$, and $\langle c \rangle$ (i.e., $\langle a(bc) \rangle$, $\langle a(bc)a \rangle$, $\langle aba \rangle$, and $\langle abc \rangle$.) They form the complete set of sequential patterns prefixed with $\langle ab \rangle$.
      iii. The $\langle (ab) \rangle$-projected database contains only two sequences: $\langle (\_c)(ac)d(cf) \rangle$ and $\langle (df)cb \rangle$, which leads to the finding of the following sequential patterns prefixed with $\langle (ab) \rangle$: $\langle c \rangle$, $\langle d \rangle$, $\langle f \rangle$, and $\langle dc \rangle$.
      iv. The $\langle ac \rangle$, $\langle ad \rangle$, and $\langle af \rangle$-projected databases can be constructed and recursively mined similarly. The sequential patterns found are shown in Table 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.
4. **The set of sequential patterns is the collection of patterns found in the above recursive mining process**. One can verify that it returns exactly the same set of sequential patterns as what *GSP* and *FreeSpan* do.

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

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

Input: A sequence database S, and the minimum support threshold min_support.

Output: The complete set of sequential patterns

Method: Call $PrefixSpan(\langle\rangle, 0, S)$.

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

Method:
1. Scan $S|_\alpha$ once, find each frequent item, b, such that
   (a) b can be assembled to the last element of α to form a sequential pattern; or
   (b) $\langle b\rangle$ can be appended to α to form a sequential pattern.
2. For each frequent item b, append it to α to form a sequential pattern α', and output α'.
3. For each α', 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 3.1 and Lemma 3.2, as shown in Theorem 3.1, later. Here, we analyze the efficiency of the algorithm as follows:

- **No candidate sequence needs to be generated by *PrefixSpan*.** Unlike a priori-like algorithms, *PrefixSpan* only grows longer sequential patterns from the shorter frequent ones. It neither generates nor tests any candidate sequence nonexistent in a projected database. Comparing with *GSP*, which generates and tests a substantial number of candidate sequences, *PrefixSpan* searches a much smaller space.
- **Projected databases keep shrinking.** As indicated in Lemma 3.2, 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 nontrivial. Techniques for reducing the number of projected databases will be discussed in the next subsection.

Theorem 3.1 (PrefixSpan). *A sequence α is a sequential pattern if and only if PrefixSpan says so.*

Proof sketch (Direction if). A length-l sequence α ($l \geq 1$) is identified as a sequential pattern by *PrefixSpan* if and only if α is a sequential pattern in the projected database of its length-$(l-1)$ prefix α^-. If $l = 1$, the length-0 prefix of α is $\alpha^- = \langle\rangle$ and the projected database is S itself. So, α is a sequential pattern in S. If $l > 1$, according to Lemma 3.2, $S|_{\alpha^-}$ is exactly the α^--projected database, and $support_S(\alpha) = support_{S|_{\alpha^-}}(\alpha)$. Therefore, if α is a sequential pattern in $S|_{\alpha^-}$, it is also a sequential pattern in S. By this, we show that a sequence α is a sequential pattern if *PrefixSpan* says so.

(Direction only-if). Lemma 3.1 guarantees that *PrefixSpan* identifies the complete set of sequential patterns in S. So, we have the theorem. □

3.3 Pseudoprojection

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 *pseudoprojection*. 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. *Pseudoprojection* 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 5 (Pseudoprojection). For the same sequence database S in Table 1 with $min\_sup = 2$, sequential patterns in S can be mined by pseudoprojection 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 of two pieces of information: 1) a *pointer* to

TABLE 3
A Sequence Database and Some of Its Pseudoprojected Databases

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

s_1 which could be the string_id s_1 and 2) the *offset(s)*, which should be a single integer, such as 2, if there is a single projection point; and a set of integers, such as $\{2, 3, 6\}$, if there are multiple projection points. Each offset indicates at which position the projection starts in the sequence.

The projected databases for prefixes $\langle a \rangle$, $\langle b \rangle$, $\langle c \rangle$, $\langle d \rangle$, $\langle f \rangle$, and $\langle aa \rangle$ are shown in Table 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 pseudoprojected database usually takes much less space than its corresponding physically projected one.

Pseudoprojection avoids physically copying suffixes. Thus, it is efficient in terms of both running time and space. However, it may not be efficient if the pseudoprojection 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 into main memory, the physical projection should be applied, however, the execution should be swapped to pseudoprojection once the projected databases can fit in main memory. This methodology is adopted in our *PrefixSpan* implementation.

Notice that the pseudoprojection 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 pseudoprojection technique for PrefixSpan.

4 Experimental Results and Performance Analysis

Since *GSP* [23] and *SPADE* [29] 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 an indepth analysis on why *PrefixSpan* outperforms the other algorithms.

4.1 Experimental Results

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 750 MHz AMD PC with 512 megabytes main memory, running Microsoft Windows 2000 Server. Three algorithms, *GSP*, *FreeSpan*, and *PrefixSpan*, were implemented by us using Microsoft Visual C++ 6.0. The implementation of the fourth algorithm, *SPADE*, is obtained directly from the author of the algorithm [29]. Detailed algorithm implementation is described as follows:

1. *GSP*. The *GSP* algorithm is implemented according to the description in [23].
2. *SPADE*. *SPADE* is tested with the implementation provided by the algorithm inventor [29].
3. *FreeSpan*. *FreeSpan* is implemented according to the algorithm described in [8].
4. *PrefixSpan*. *PrefixSpan* is implemented as described in this paper,[3] with pseudoprojection turned on in most cases. Only in the case when testing the role of pseudoprojection, two options are adopted: one with the pseudoprojection function *turned on* and the other with it *turned off*.

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

For real data set, we have obtained the *Gazelle* data set from Blue Martini. This data set has been used in KDD-CUP'2000 and contains a total of 29,369 customers' Web click-stream data provided by Blue Martini Software company. For each customer, there may be several sessions of Web click-stream and each session can have multiple page views. Because each session is associated with both starting and ending date/time, for each customer, we can sort its sessions of click-stream into a sequence of page views according to the viewing date/time. This data set contains 29,369 sequences (i.e., customers), 35,722 sessions (i.e., transactions or events), and 87,546 page views (i.e.,

3. Note that the previous version of the *PrefixSpan* algorithm published in [19] introduces another optimization technique called *bilevel projection* which performs physical database projection at every two levels. Based on our recent substantial performance study, the role of bilevel projection in performance improvement is only marginal in certain cases, but can barely offset its overhead in many other cases. Thus, this technique is dropped from the *PrefixSpan* options and also from the performance study.

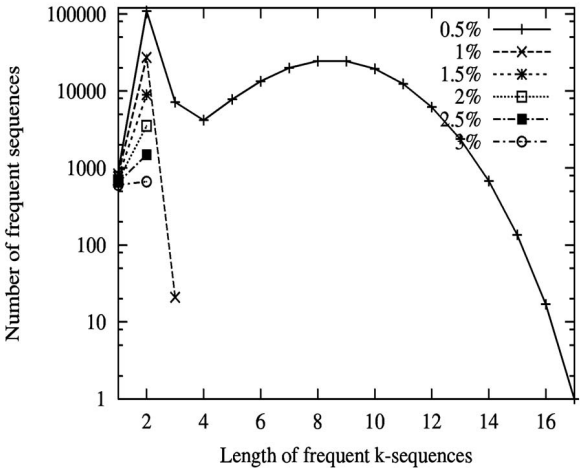


Fig. 2. Distribution of frequent sequences of data set $C10T8S8I8$.

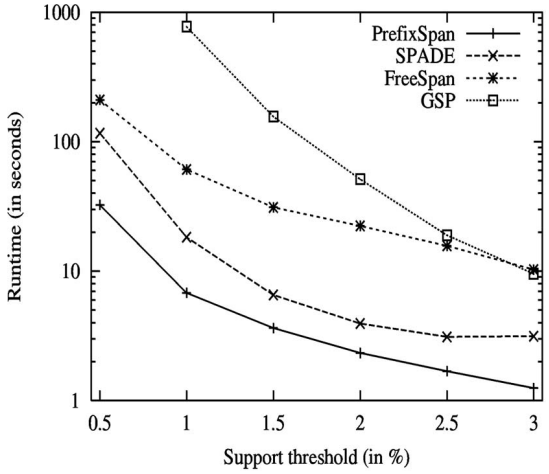


Fig. 3. Performance of the four algorithms on data set $C10T8S8I8$.

products or items). There are in total 1,423 distinct page views. More detailed information about this data set can be found in [10].

For synthetic data sets, we have also used a large set of synthetic sequence data generated by a data generator similar in spirit to the IBM data generator [2] designed for testing sequential pattern mining algorithms. Various kinds of sizes and data distributions of data sets are generated and tested in this performance study. The convention for the data sets is as follows: $C200T2.5S10I1.25$ means that the data set contains $200k$ customers (i.e., sequences) and the number of items is 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 four 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 [29]. 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 four transactions, and each transaction is composed of eight items. Fig. 2 shows the distribution of frequent sequences of data set $C10T8S8I8$, from which one can see that when *min_support* is no less than 1 percent, 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. 3 shows the processing time of the four algorithms at different support thresholds. The processing times are sorted in time ascending order as "*PrefixSpan* $<$ *SPADE* $<$ *FreeSpan* $<$ GSP". When $min\_support = 1\%$, *PrefixSpan* (runtime $=$ 6.8 seconds) is about two orders of magnitude faster than *GSP* (runtime $=$ 772.72 seconds). When *min_support* is reduced to 0.5 percent, the data set contains a large number of frequent sequences, *PrefixSpan* takes 32.56 seconds, which is more than 3.5 times faster than *SPADE* (116.35 seconds), while *GSP* never terminates on our machine.

The second test is performed on the data set $C200T2.5S10I1.25$, which is much larger than the first one since it contains $200k$ customers (i.e., sequences) and the number of items is 10,000. However, it is sparser than the first data set since 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 four transactions, and each transaction is composed of 1.25 items. Fig. 4 shows the distribution of frequent sequences of the data set, from which one can see that the number of longer frequent sequences is almost always smaller than that of shorter ones (except when $min\_support = 0.25\%$). Fig. 5 shows the processing time of the four algorithms at different support thresholds. The processing time maintains the same order "*PrefixSpan* $<$ *SPADE* $<$ *FreeSpan* $<$ *GSP*" when *min_support* is small. When $min\_support =$ 0.5-0.75 percent, *GSP*, *SPADE*, and *FreeSpan* have very similar running time (but *PrefixSpan* is still 2-3 times faster).

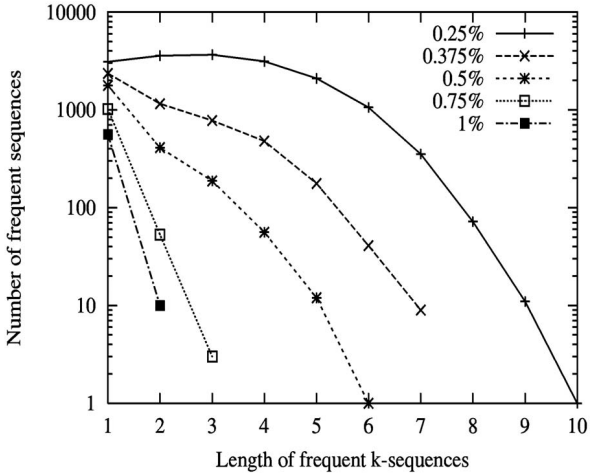


Fig. 4. Distribution of frequent sequences of data set $C200T2.5S10I1.25$.

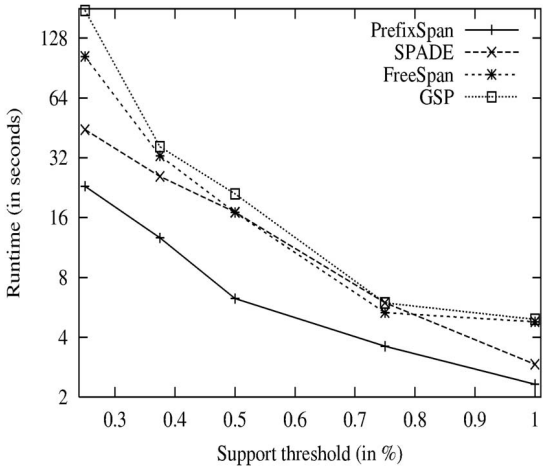


Fig. 5. Performance of the four algorithms on data set $C200T2.5S10I1.25$.

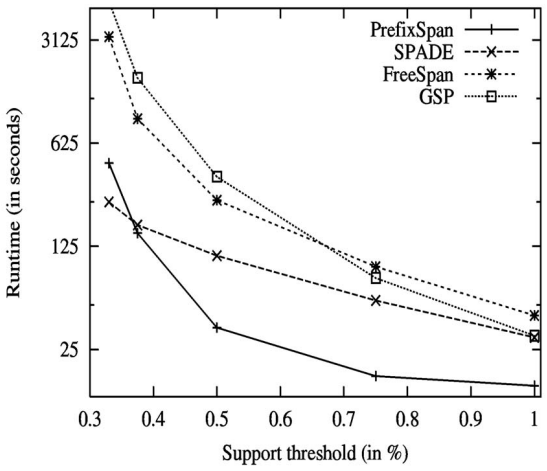


Fig. 7. Performance of the four algorithms on data set $C200T5S10I2.5$.

The third test is performed on the data set $C200T5S10I2.5$, which is substantially denser than the second data set, although it still contains $200k$ customers (i.e., sequences) and the number of items is 10,000. This is because the average number of items in a transaction (i.e., event) is five, the average number of transactions in a sequence is 10, and, on average, a frequent sequential pattern consists of four transactions, and each transaction is composed of 2.5 items. Fig. 6 shows the distribution of frequent sequences of the data set, from which one can see that the number of longer frequent sequences grows into nontrivial range. Fig. 7 shows the processing time of the four algorithms at different support thresholds. The testing result makes clear distinction among the algorithms tested. It shows almost the same ordering of the algorithms for running time, "*PrefixSpan* $<$ *SPADE* $<$ *FreeSpan* $<$ *GSP*", except *SPADE* is slightly faster than *PrefixSpan* when $min\_support = 0.33\%$. However, when $min\_support =$ 0.25 percent, the number of frequent sequences grows to more than 4 million, and in this case, only *PrefixSpan* can run it (with running time = 3539.78 seconds) and all the other three algorithms cannot finish (and, thus, not shown in the figure).

The performance study on the real data set *Gazelle* is reported as follows: Fig. 8 shows the distribution of frequent sequences of Gazelle data set for different support thresholds. We can see that this data set is a very sparse data set: Only when the support threshold is lower than 0.05 percent are there some long frequent sequences. Fig. 9 shows the performance comparison among the four algorithms for Gazelle data set. 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 percent, but once the support threshold is no greater than 0.018 percent, it cannot stop running.

Figs. 10, 11, and 12 show the results of scalability tests of the four algorithms on data set $T2.5S10I1.25$, with the database size growing from 200K to 1,000K sequences, and with different *min_support* threshold settings. With $min\_support = 1$ percent, all the algorithms terminate within 26 seconds, even with the database of 1,000K sequences, and *PrefixSpan* has the best performance overall but *GSP* is actually marginally better when the database size is between 600K and 800K sequences. However, when the

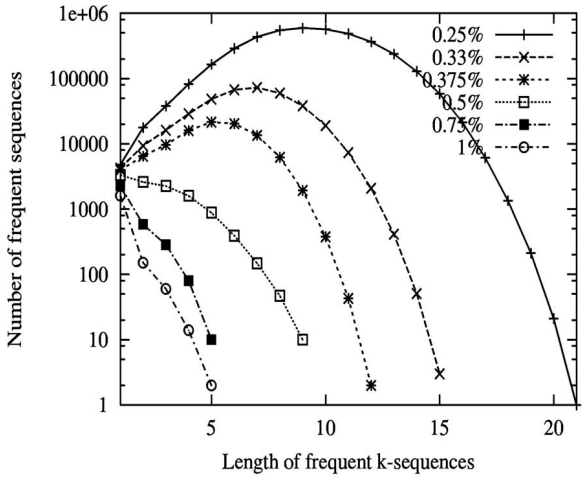


Fig. 6. Distribution of frequent sequences of data set $C200T5S10I2.5$.

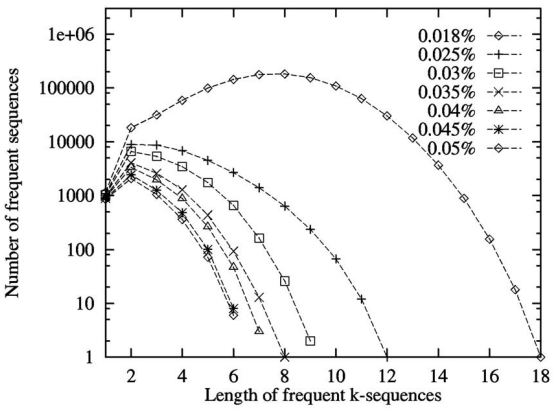


Fig. 8. Distribution of frequent sequences of data set Gazelle.

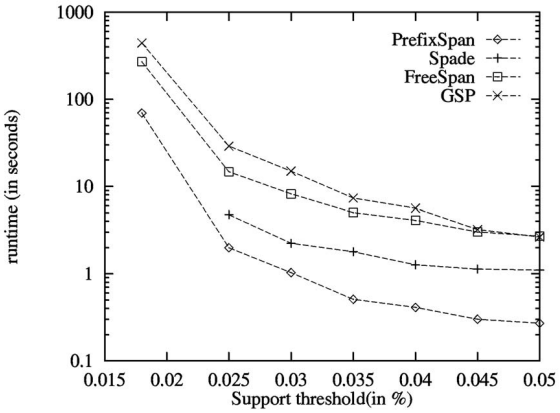


Fig. 9. Performance of the four algorithms on data set Gazelle.

min_support drops down to 0.25 percent, *PrefixSpan* has the clear competitive edge. For the database of 1,000K sequences, *PrefixSpan* is about nine times faster than *GSP*, but *SPADE* cannot terminate.

To test the effectiveness of pseudoprojection, a performance test was conducted to compare three algorithms:

1. *PrefixSpan* with the pseudoprojection function turned on,
2. *PrefixSpan* with pseudoprojection turned off, and
3. *SPADE*.

The test data is the same data set $C10T8S8I8$, used in the experiment of Fig. 3. However, the *min_support* threshold is set low at the range of 0.25 percent to 0.5 percent. Notice that the previous test was between 0.5 percent and 3 percent, which was the range that *FreeSpan* and *GSP* can still run within a reasonable amount of time, although *GSP* cannot run when *min_support* is reduced down to 0.5 percent. The distribution of frequent sequences of data set $C10T8S8I8$, with very low *min_support* is shown in Fig. 13. It shows that the total number of frequent patterns grows up to around 4 million frequent sequences when $min\_support = 0.25\%$. The testing result is shown in Fig. 14, which demonstrates that

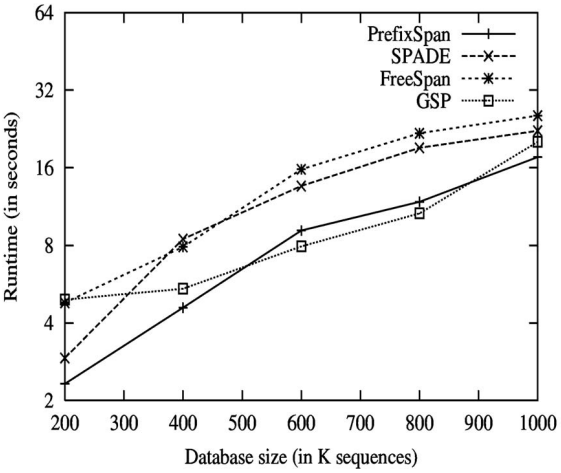


Fig. 10. Scalability test of the four algorithms on data set $T2.5S10I1.25$, with min_support 1 percent.

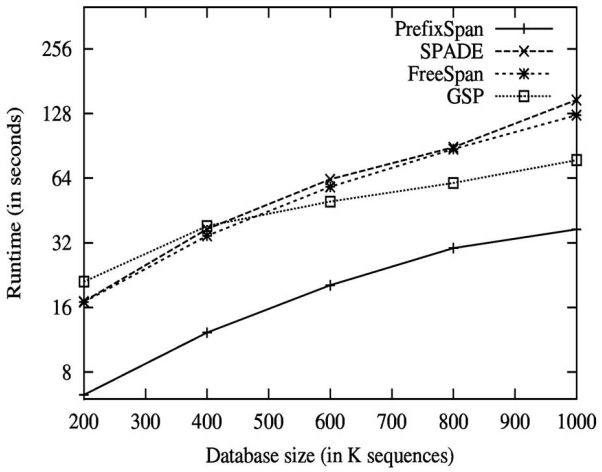


Fig. 11. Scalability test of the four algorithms on data set $T2.5S10I1.25$, with min_support 0.5 percent.

PrefixSpan with pseudoprojection is consistently faster than that without pseudoprojection although the gap is not big in most cases, but is about 40 percent performance improvement when $min\_support = 0.25\%$. Moreover, the performance of *SPADE* is clearly behind *PrefixSpan* when *min_support* is very low and the sequential patterns are dense.

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. 15 shows the results for Gazelle data set, 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 percent, *GSP* consumes about 40 MB memory and *SPADE* just cannot stop running after it has used more than 22 MB memory while *PrefixSpan* only uses about 2.7 MB memory.

Fig. 16 demonstrates the memory usage for data set 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 percent, *GSP* cannot stop running after it has consumed about 362 MB

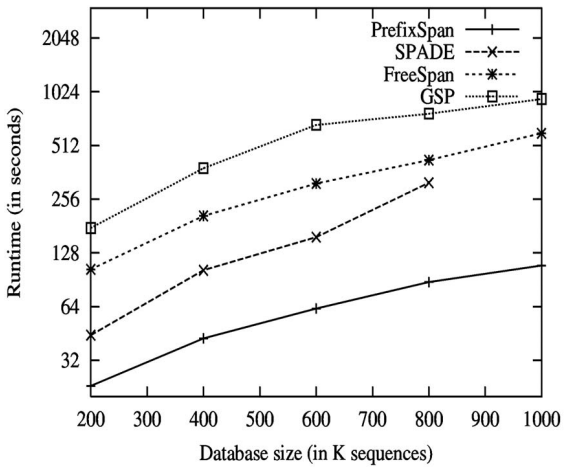


Fig. 12. Scalability test of the four algorithms on data set $T2.5S10I1.25$, with min_support 0.25 percent.

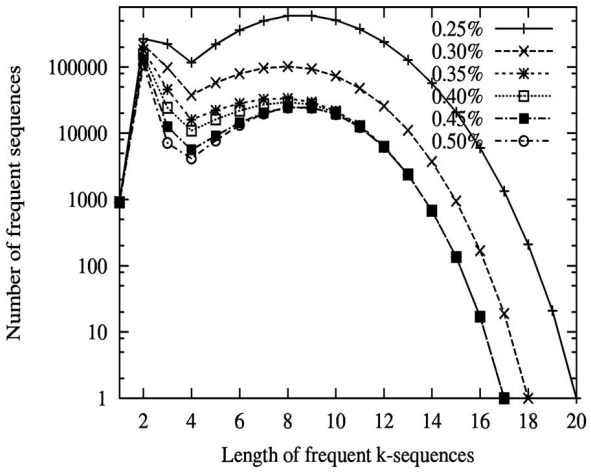


Fig. 13. Distribution of frequent sequences of data set $C10T8S8I8$, with very low min_support (ranging from 0.25 percent to 0.5 percent).

memory and *SPADE* reported an error message, "*memory:: Array: Not enough memory*," when it tried to allocate another bulk of memory after it has used about 262 MB memory, while *PrefixSpan* only uses 108 MB memory. This also explains why in several cases in our previous experiments when the support threshold becomes really low, only *PrefixSpan* can finish running.

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

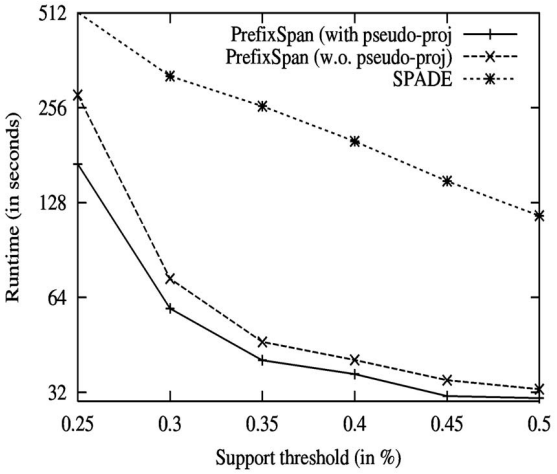


Fig. 14. Performance of *PrefixSpan*(with versus without pseudoprojection) versus SPADE on data $C10T8S8I8$, with very low min_support.

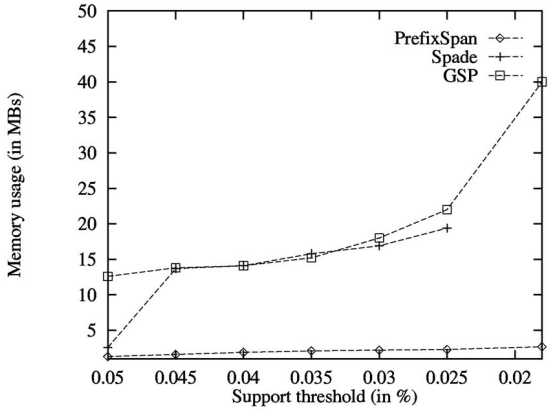


Fig 15. Memory usage comparison among *PrefixSpan*, *SPADE*, and *GSP* for data set Gazelle.

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 [29]. *GSP* performs fairly well only when min_support is rather high, with good scalability, which is consistent with the performance study reported in [23]. 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 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 (as shown in Fig. 7). This leaves some room for performance improvement in the future.

4.2 Why Does *PrefixSpan* Have High Performance?

With the above comprehensive performance study, we are convinced that *PrefixSpan* is the clear winner among all the four tested algorithms. The question becomes why *PrefixSpan* has such high performance: Is it because of some implementation tricks, or is it inherently in the algorithm itself? We have the following analysis:

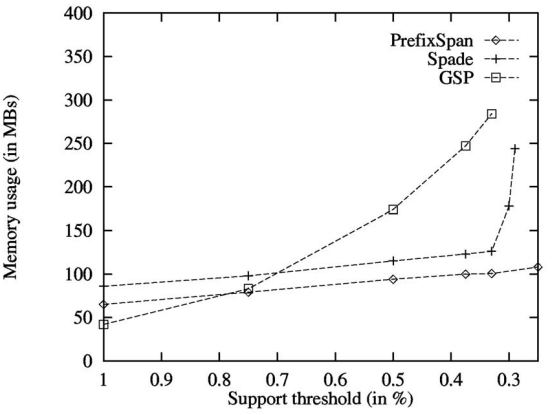


Fig. 16. Memory usage: *PrefixSpan*, *SPADE*, and *GSP* for synthetic data set C200T5S10I2.5.

- **Pattern-growth without candidate generation.** *PrefixSpan* is a pattern growth-based approach, similar to *FP-growth* [9]. Unlike traditional a priori-based approach which performs candidate generation-and test, *PrefixSpan* does not generate any useless candidate and it only counts the frequency of local 1-itemsets. Our performance shows that when *min_support* drops, the number of frequent sequences grows up exponentially. As a candidate generation-and-test approach, like *GSP*, handling such exponential number of candidates is the must, independent of optimization tricks. It is no wonder that *GSP* costs an exponential growth amount of time to process a pretty small database (e.g., only $10K$ sequences) since it must take a great deal of time to generate and test a huge number of sequential pattern candidates.
- **Projection-based divide-and-conquer as an effective means for data reduction.** *PrefixSpan* grows longer patterns from shorter ones by dividing the search space and focusing only on the subspace potentially supporting further pattern growth. The search space of *PrefixSpan* is focused and is confined to a set of projected databases. Since a projected database for a sequential pattern α contains all and only the necessary information for mining the sequential patterns that can grow from α, the size of the projected databases usually reduces quickly as mining proceeds to longer sequential patterns. In contrast, the a priori-based approach always searches the original database at each iteration. Many irrelevant sequences have to be scanned and checked, which adds to the overhead. This argument is also supported by our performance study.
- ***PrefixSpan* consumes relatively stable memory space** because it generates no candidates and explores the divide-and-conquer methodology. On the other hand, the candidate generation-and-test methods, including both *GSP* and *SPADE*, require a substantial amount of memory when the support threshold goes low since it needs to hold a tremendous number of candidate sets.
- ***PrefixSpan* applies prefix-projected pattern growth which is more efficient than *FreeSpan* that uses frequent pattern-guided projection.** Comparing with frequent pattern-guided projection, employed in *FreeSpan*, prefix-projected pattern growth (*PrefixSpan*) saves a lot of time and space because *PrefixSpan* projects only on the frequent items in the suffixes, and the projected sequences shrink rapidly. When mining in dense databases where *FreeSpan* cannot gain much from the projections, *PrefixSpan* can still reduce substantially both the length of sequences and the number of sequences in projected databases.

5 Extensions and Discussions

Comparing with mining (unordered) frequent patterns, mining sequential patterns is one step toward mining more sophisticated frequent patterns in large databases. With the successful development of pattern-growth-based sequential pattern mining method, such as *PrefixSpan*, it is interesting to explore how such a method can be extended to handle more sophisticated cases. It is straightforward to extend this approach to mining multidimensional, multilevel sequential patterns [21]. In this section, we will discuss constraint-based mining of sequential patterns and a few research problems.

5.1 Constraint-Based Mining of Sequential Patterns

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

Constraint-based mining has been studied extensively in frequent pattern mining, such as [15], [4], [18]. In general, constraints can be characterized based on the notion of monotonicity, antimonotonicity, 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 [18]. 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 [6], can be handled elegantly. Let us examine one such example.

Example 6 (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 antimonotone, nor monotone, nor succinct, the classical constraint-pushing framework [15] cannot push it deep. To overcome this difficulty, Guha et al. [6] 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 [23], a typical candidate generation-and-test approach.

With the development of the pattern-growth methodology, such kinds of constraints can be pushed deep easily and elegantly into the sequential pattern mining process [20]. This is because, in the context of *PrefixSpan*, a regular expression constraint has a nice property called **growth-based antimonotonic**. A constraint is *growth-based antimonotonic* 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 6 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 [20].

5.2 Problems for Further Research

Although the sequential pattern growth approach proposed in this paper is efficient and scalable, there are still some challenging research issues with regards to sequential pattern mining, especially for certain large scale applications. Here, we illustrate a few problems that need further research.

5.2.1 Mining Closed and Maximal Sequential Patterns

A frequent long sequence contains a combinatorial number of frequent subsequences, as shown in Section 1. For a sequential pattern of length 100, there exist $2^{100} - 1$ nonempty subsequences. In such cases, it is prohibitively expensive to mine the complete set of patterns no matter which method is to be applied.

Similar to mining closed and maximal frequent patterns in transaction databases [17], [3], which mines only the longest frequent patterns (in the case of max-pattern mining) or the longest one with the same support (in the case of closed-pattern mining), for sequential pattern mining, it is also desirable to mine only (frequent) *maximal* or *closed* sequential patterns, where a sequence s is *maximal* if there exists no frequent supersequence of s, while a sequence s is *closed* if there exists no supersequence of s with the same support as s.

The development of efficient algorithms for mining closed and maximal sequential patterns in large databases is an important research problem. A recent study in [27] proposed an efficient closed sequential pattern method, called *CloSpan*, as a further development of the *PrefixSpan* mining framework, influenced by this approach.

5.2.2 Mining Approximate Sequential Patterns

In this study, we have assumed all the sequential patterns to be mined are exact matching patterns. In practice, there are many applications that need approximate matches, such as DNA sequence analysis which allows limited insertions, deletions, and mutations in their sequential patterns. The development of efficient and scalable algorithms for mining approximate sequential patterns is a challenging and practically useful direction to pursue. A recent study on mining long sequential patterns in a noisy environment [28] is a good example in this direction.

5.2.3 Toward Mining Other Kinds of Structured Patterns

Besides mining sequential patterns, another important task is the mining of frequent substructures in a database composed of structured or semistructured data sets. The substructures may consist of trees, directed-acyclic graphs (i.e., DAGs), or general graphs which may contain cycles. There are a lot of applications related to mining frequent substructures since most human activities and natural processes may contain certain structures, and a huge amount of such data has been collected in large data/information repositories, such as molecule or biochemical structures, Web connection structures, and so on. It is important to develop scalable and flexible methods for mining structured patterns in such databases. There have been some recent work on mining frequent subtrees, such as [31], and frequent subgraphs, such as [11], [25] in structured databases, where [25] shows that the pattern growth approach has clear performance edge over a candidate generation-and-test approach. Furthermore, as discussed above, is it is more desirable to mine closed frequent subgraphs (a subgraph g is *closed* if there exists no supergraph of g carrying the same support as g) than mining explicitly the complete set of frequent subgraphs because a large graph inherently contains an exponential number of subgraphs. A recent study [26] has developed an efficient closed subgraph pattern method, called *CloseGraph*, which is also based on the pattern-growth framework and influenced by this approach.

6 Conclusions

We have performed a systematic study on mining of sequential patterns in large databases and developed a *pattern-growth approach* for efficient and scalable mining of sequential patterns.

Instead of refinement of the a priori-like, candidate generation-and-test approach, such as *GSP* [23], we promote a divide-and-conquer approach, called *pattern-growth approach*, which is an extension of *FP-growth* [9], an efficient pattern-growth algorithm for mining frequent patterns without candidate generation.

An efficient pattern-growth method, *PrefixSpan*, is proposed and studied in this paper. *PrefixSpan* recursively projects a sequence database into a set of smaller projected sequence databases and grows sequential patterns in each projected database by exploring only locally frequent fragments. It mines the complete set of sequential patterns and substantially reduces the efforts of candidate subsequence generation. Since *PrefixSpan* explores ordered growth by prefix-ordered expansion, it results in less "growth points" and reduced projected databases in comparison with our previously proposed pattern-growth algorithm, *FreeSpan*. Furthermore, a *pseudoprojection* technique is proposed for *PrefixSpan* to reduce the number of physical projected databases to be generated. A comprehensive performance study shows that *PrefixSpan* outperforms the a priori-based *GSP* algorithm, *FreeSpan*, and *SPADE* in most cases, and *PrefixSpan* integrated with pseudoprojection is the fastest among all the tested algorithms.

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 multilevel, multidimensional sequential patterns, mining sequential patterns with user-specified constraints, and so on. 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, such as mining closed and maximal sequential patterns, mining approximate sequential patterns, and extension of the method toward mining structured patterns. 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.

ACKNOWLEDGMENTS

The authors are grateful to Dr. Mohammed Zaki for providing them with the source code of *SPADE* and the vertical data conversion package, as well as promptly answering many of their questions related to *SPADE*. Also, they would like to express their thanks to Blue Martini Software Inc. for sending them the *Gazelle* data set and allowing them to publish their algorithm performance results using this data set. 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 US National Science Foundation NSF IIS-02-09199, NSF IIS-03-08001, and the University of Illinois. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies. This paper is a major-value added version of the following conference paper [19].

REFERENCES

[1] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," *Proc. 1994 Int'l Conf. Very Large Data Bases (VLDB '94),* pp. 487-499, Sept. 1994.

[2] R. Agrawal and R. Srikant, "Mining Sequential Patterns," *Proc. 1995 Int'l Conf. Data Eng. (ICDE '95),* pp. 3-14, Mar. 1995.

[3] R.J. Bayardo, "Efficiently Mining Long Patterns from Databases," *Proc. 1998 ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD '98),* pp. 85-93, June 1998.

[4] R.J. Bayardo, R. Agrawal, and D. Gunopulos, "Constraint-Based Rule Mining on Large, Dense Data Sets," *Proc. 1999 Int'l Conf. Data Eng. (ICDE '99),* pp. 188-197, Apr. 1999.

[5] C. Bettini, X.S. Wang, and S. Jajodia, "Mining Temporal Relationships with Multiple Granularities in Time Sequences," *Data Eng. Bull.,* vol. 21, pp. 32-38, 1998.

[6] S. Guha, R. Rastogi, and K. Shim, "Rock: A Robust Clustering Algorithm for Categorical Attributes," *Proc. 1999 Int'l Conf. Data Eng. (ICDE '99),* pp. 512-521, Mar. 1999.

[7] J. Han, G. Dong, and Y. Yin, "Efficient Mining of Partial Periodic Patterns in Time Series Database," *Proc. 1999 Int'l Conf. Data Eng. (ICDE '99),* pp. 106-115, Apr. 1999.

[8] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu, "FreeSpan: Frequent Pattern-Projected Sequential Pattern Mining," *Proc. 2000 ACM SIGKDD Int'l Conf. Knowledge Discovery in Databases (KDD '00),* pp. 355-359, Aug. 2000.

[9] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns without Candidate Generation," *Proc. 2000 ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD '00),* pp. 1-12, May 2000.

[10] R. Kohavi, C. Brodley, B. Frasca, L. Mason, and Z. Zheng, "KDD-Cup 2000 Organizers' Report: Peeling the Onion," *Proc. SIGKDD Explorations,* vol. 2, pp. 86-98, 2000.

[11] M. Kuramochi and G. Karypis, "Frequent Subgraph Discovery," *Proc. 2001 Int'l Conf. Data Mining (ICDM '01),* pp. 313-320, Nov. 2001.

[12] H. Lu, J. Han, and L. Feng, "Stock Movement and n-Dimensional Inter-Transaction Association Rules," *Proc. 1998 SIGMOD Workshop Research Issues on Data Mining and Knowledge Discovery (DMKD '98),* vol. 12, pp. 1-7, June 1998.

[13] H. Mannila, H Toivonen, and A.I. Verkamo, "Discovery of Frequent Episodes in Event Sequences," *Data Mining and Knowledge Discovery,* vol. 1, pp. 259-289, 1997.

[14] F. Masseglia, F. Cathala, and P. Poncelet, "The PSP Approach for Mining Sequential Patterns," *Proc. 1998 European Symp. Principle of Data Mining and Knowledge Discovery (PKDD '98),* pp. 176-184, Sept. 1998.

[15] R. Ng, L.V.S. Lakshmanan, J. Han, and A. Pang, "Exploratory Mining and Pruning Optimizations of Constrained Associations Rules," *Proc. 1998 ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD '98),* pp. 13-24, June 1998.

[16] B. Özden, S. Ramaswamy, and A. Silberschatz, "Cyclic Association Rules," *Proc. 1998 Int'l Conf. Data Eng. (ICDE '98),* pp. 412-421, Feb. 1998.

[17] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal, "Discovering Frequent Closed Itemsets for Association Rules," *Proc. Seventh Int'l Conf. Database Theory (ICDT '99),* pp. 398-416, Jan. 1999.

[18] J. Pei, J. Han, and L.V.S. Lakshmanan, "Mining Frequent Itemsets with Convertible Constraints," *Proc. 2001 Int'l Conf. Data Eng. (ICDE '01),* pp. 433-332, Apr. 2001.

[19] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu, "PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth," *Proc. 2001 Int'l Conf. Data Eng. (ICDE '01),* pp. 215-224, Apr. 2001.

[20] J. Pei, J. Han, and W. Wang, "Constraint-Based Sequential Pattern Mining in Large Databases," *Proc. 2002 Int'l Conf. Information and Knowledge Management (CIKM '02),* pp. 18-25, Nov. 2002.

[21] H. Pinto, J. Han, J. Pei, K. Wang, Q. Chen, and U. Dayal, "Multi-Dimensional Sequential Pattern Mining," *Proc. 2001 Int'l Conf. Information and Knowledge Management (CIKM '01),* pp. 81-88, Nov. 2001.

[22] S. Ramaswamy, S. Mahajan, and A. Silberschatz, "On the Discovery of Interesting Patterns in Association Rules," *Proc. 1998 Int'l Conf. Very Large Data Bases (VLDB '98),* pp. 368-379, Aug. 1998.

[23] R. Srikant and R. Agrawal, "Mining Sequential Patterns: Generalizations and Performance Improvements," *Proc. Fifth Int'l Conf. Extending Database Technology (EDBT '96),* pp. 3-17, Mar. 1996.

[24] J. Wang, G. Chirn, T. Marr, B. Shapiro, D. Shasha, and K. Zhang, "Combinatiorial Pattern Discovery for Scientific Data: Some Preliminary Results," *Proc. 1994 ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD '94),* pp. 115-125, May 1994.

[25] X. Yan and J. Han, "gSpan: Graph-Based Substructure Pattern Mining," *Proc. 2002 Int'l Conf. Data Mining (ICDM '02),* pp. 721-724, Dec. 2002.

[26] X. Yan and J. Han, "CloseGraph: Mining Closed Frequent Graph Patterns," *Proc. 2003 ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining (KDD '03),* Aug. 2003.

[27] X. Yan, J. Han, and R. Afshar, "CloSpan: Mining Closed Sequential Patterns in Large Datasets," *Proc. 2003 SIAM Int'l Conf. Data Mining (SDM '03),* pp. 166-177, May 2003.

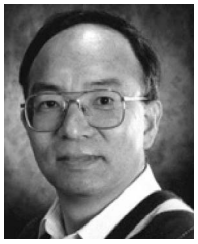
[28] J. Yang, W. Wang, P.S. Yu, and J. Han, "Mining Long Sequential Patterns in a Noisy Environment," *Proc. 2002 ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD '02),* pp. 406-417, June 2002.

[29] M. Zaki, "SPADE: An Efficient Algorithm for Mining Frequent Sequences," *Machine Learning,* vol. 40, pp. 31-60, 2001.

[30] M.J. Zaki, "Efficient Enumeration of Frequent Sequences," *Proc. Seventh Int'l Conf. Information and Knowledge Management (CIKM '98),* pp. 68-75, Nov. 1998.

[31] M.J. Zaki, "Efficiently Mining Frequent Trees in a Forest," *Proc. 2002 ACM SIGKDD Int'l Conf. Knowledge Discovery in Databases (KDD '02),* pp. 71-80, July 2002.

Jian Pei received the BEng and MEng degrees, both in computer science from Shanghai Jiao Tong University, China, in 1991 and 1993, respectively, and the PhD degree in computing science from Simon Fraser University, Canada, in 2002. He was a PhD candidate at Peking University from 1997-1999. He is currently an assistant professor of computing science at Simon Fraser University, Canada. His research interests include data mining, data warehousing, online analytical processing, database systems, and bioinformatics. His research is currently supported in part by the US National Science Foundation (NSF). He has served on the program committees of international conferences and workshops, and has 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. He is a guest area editor of the *Journal of Computer Science and Technology*.



Jiawei Han received the PhD degree in computer science from the University of Wisconsin in 1985. He is a professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign. Previously, he was an Endowed University Professor at Simon Fraser University, Canada. He has been working on research into data mining, data warehousing, spatial and multimedia databases, deductive and object-oriented databases, and biomedical databases, with more than 250 journal and conference publications. He has chaired or served in many program committees of international conferences and workshops, including ACM SIGKDD conferences (2001 best paper award chair, 2002 student award chair, 1996 PC cochair), SIAM-Data Mining conferences (2001 and 2002 PC cochair), ACM SIGMOD Conferences (2000 exhibit program chair), and International Conferences on Data Engineering (2004 and 2002 PC vicechair). He also served or is serving on the editorial boards for *Data Mining and Knowledge Discovery: An International Journal*, *IEEE Transactions on Knowledge and Data Engineering*, and the *Journal of Intelligent Information Systems*. 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). Dr. Han has received IBM Faculty Award, the Outstanding Contribution Award at the 2002 International Conference on Data Mining, and the ACM Service Award. He is the first author of the textbook *Data Mining: Concepts and Techniques* (Morgan Kaufmann, 2001). He is a senior member of the IEEE.

Behzad Mortazavi-Asl received the BSc and MSc degrees in computing science from Simon Fraser University in 1998 and 2001, respectively. His research interests include WebLog mining, sequential pattern mining, and online educational systems. He is currently working as a software developer.

Jianyong Wang received the PhD degree in computer science in 1999 from the Institute of Computing Technology, the Chinese Academy of Sciences. Since then, he has worked as an assistant professor in the Department of Computer Science and Technology, Peking (Beijing) University in the areas of distributed systems and Web search engines, and visited the School of Computing Science at Simon Fraser University and the Department of Computer Science at the University of Illinois at Urbana-Champaign as a postdoc research fellow, mainly working in the area of data mining. He is currently a research associate of the Digital Technology Center at the University of Minnesota.

Helen Pinto received the MSc degree in computing science in 2001 from Simon Fraser University, Canada. She is currently with the Alberta Research Council in Calgary, Alberta, Canada. Since 2001, she has been working in the area of intelligent systems applications with emphasis on knowledge discovery in databases.

Qiming Chen is a senior technical staff in PacketMotion Inc. Prior to that he worked at HP Labs, Hewlett Packard Company as a senior computer scientist, and at Commerce One Labs, Commerce One Inc. as a principal architect. His current research and development activities are in the areas of Grid services, Web services, interbusiness process and conversation management, as well as data warehousing, OLAP, and data mining. His early research interests include database, workflow, and distributed and agent systems. He authored more than 80 technical publications and has a number of patents granted or filed. He has served in the program committees of more than 20 international conferences.

Umeshwar Dayal received the PhD degree in applied mathematics from Harvard University. He is Director of the Intelligent Enterprise Technologies Laboratory at Hewlett-Packard Laboratories. His research interests are data mining, business process management, distributed information management, and decision support technologies, especially as applied to e-business. Dr. Dayal is a member of the ACM and the IEEE Computer Society.

Mei-Chun Hsu received the BA degree from the National Taiwan University, the MS degree from the University of Massachusetts at Amherst, and the PhD degree from the Massachusetts Institute of Technology. She is currently vice president of engineering at Commerce One Inc. Dr. Hsu has led the design and development efforts for the Commerce One Conductor 6.0, a web service-enabled application integration platform. Prior to Commerce One, she held positions with Hewlett-Packard Laboratories, EDS, and Digital Equipment Corporation. She was also a member of Computer Science Faculty at Harvard University. Her research interests span database systems, transaction processing, business process management systems, and data mining technologies.