Knowl Inf Syst (2008) 15:149–179
DOI 10.1007/s10115-006-0056-0

Knowledge and Information Systems

REGULAR PAPER

Congnan Luo · Soon M. Chung

A scalable algorithm for mining maximal frequent sequences using a sample

Received: 14 April 2005 / Revised: 15 September 2006 / Accepted: 7 October 2006 /
Published online: 24 January 2007

Abstract In this paper, we propose an efficient scalable algorithm for mining *M*aximal *S*equential *P*atterns using *S*ampling (MSPS). The MSPS algorithm reduces much more search space than other algorithms because both the subsequence infrequency-based pruning and the supersequence frequency-based pruning are applied. In MSPS, a sampling technique is used to identify long frequent sequences earlier, instead of enumerating all their subsequences. We propose how to adjust the user-specified minimum support level for mining a sample of the database to achieve better overall performance. This method makes sampling more efficient when the minimum support is small. A signature-based method and a hash-based method are developed for the subsequence infrequency-based pruning when the seed set of frequent sequences for the candidate generation is too big to be loaded into memory. A prefix tree structure is developed to count the candidate sequences of different sizes during the database scanning, and it also facilitates the customer sequence trimming. Our experiments showed MSPS has very good performance and better scalability than other algorithms.

Keywords Data mining · Maximal frequent sequences · Sampling · Signatures · Prefix tree · Performance analysis

1 Introduction

Mining sequential patterns from large databases is an important problem in data mining. With numerous practical applications, such as consumer market-basket data analysis and Web-log analysis, it has become an active research topic. Since it

C. Luo · S. M. Chung (✉)
Department of Computer Science and Engineering,
Wright State University, Dayton, OH 45435, USA
E-mail: {luo.3, soon.chung}@wright.edu

was introduced in Agrawal and Srikant [3], many algorithms have been proposed, but most of them are to discover the full set of frequent sequences.

In pure bottom-up, breadth-first search algorithms such as GSP [17] and PSP [12], only subsequence infrequency-based pruning is used to reduce the number of candidate sequences. So, if a sequence with length l is frequent, all of its 2^l subsequences must be enumerated first. Thus, if some frequent sequences are long, the overhead of enumerating all of their subsequences is so much that mining the full set of frequent sequences is impractical. An alternative approach is mining only the maximal frequent sequences. A frequent sequence is maximal if none of its supersequences is frequent. Mining only the maximal frequent sequences is efficient because the search space can be reduced a lot by using the supersequence frequency-based pruning. Another approach is to mine only the closed frequent sequences [19, 20]. A frequent sequence is closed if none of its supersequences has the same support. Obviously, the relationship between the set of all frequent sequences (FS), the set of closed frequent sequences (CFS), and the set of maximal frequent sequences (MFS) is MFS $\subseteq$ CFS $\subseteq$ FS. An advantage of mining CFS is that the count information for all frequent sequences is also obtained. However, in many cases, CFS could be still too large and orders of magnitude larger than MFS. Thus, mining MFS can be most efficient and scalable. In interactive data mining, after mining MFS quickly, we can selectively count the interesting patterns subsumed by MFS by scanning the database just once. Moreover, managing and querying a small set of maximal patterns is easy, time-saving, and space-saving.

The main challenge in mining MFS is how to look ahead for long or maximal frequent sequences at a reasonable cost. If the look-ahead is not cost-effective, its cost can offset the gain from the supersequence frequency-based pruning, like the cases of AprioriSome and DynamicSome algorithms [3].

Although there are many maximal frequent itemset mining algorithms [9], they cannot efficiently mine the maximal frequent sequences because of the unique characteristics of the sequence mining. For example, an item can appear multiple times in a sequence at different positions. Thus, the search space becomes much larger. We consider the look-ahead technique used in Max-Miner [5], DepthProject [1], and MAFIA [6] algorithms to find potential maximal frequent itemsets. They use a Lexicographic tree of itemsets to represent the search space, where each node is associated with a frequent itemset, called *head*, and a set of extension items, called *tail*. In this case, the union, $head \cup tail$, is the only one candidate maximal itemset to be checked for the node. But in the case of maximal frequent sequence mining, the number of candidate maximal sequences to be checked for the node is unlimited because each item in the tail can be included many times in a candidate. For example, if the head contains only one item A and the tail contains items B and C for a node, then possible candidate maximal sequences could be $A-B-C$, $A-B-C-C$, $A-B-C-C-C$, and so on. Thus, the look-ahead method of those maximal frequent itemset mining algorithms cannot work well for the mining of maximal frequent sequences.

Some sequence mining algorithms, like SPAM [4], perform a depth-first traversal of the Lexicographic sequence tree to mine long patterns. But they cannot use the subsequence infrequency-based pruning effectively because the information about infrequent short patterns is not enough. This is not so serious in the case

of maximal frequent itemset mining, like DepthProject [1], because the search space is not usually very large. But, it could be a problem in sequence mining.

In this research, we explored the sampling technique to tackle the key issue of cost-effective look-ahead in mining maximal frequent sequences. With sampling, we can combine the Apriori candidate generation method [2, 3, 17] and the supersequence frequency-based pruning. This enables us to avoid counting most nonmaximal patterns against the whole database, while finding the maximal patterns quickly.

The main search strategy of our MSPS algorithm is bottom-up and breadth-first. But after the pass 2 over the database, we mine a small random sample database first, starting with the candidate 3-sequences (i.e., sequences of three items) generated from the set of frequent 2-sequences. The local maximal frequent sequences that are found from the sample database starting with the global candidate 3-sequences, are verified in a top-down fashion against the original database, so that we can efficiently collect the longest frequent sequences covered by them. Then, the bottom-up search is resumed from the pass 3, and the supersequence frequency-based pruning is applied at each pass by using the long patterns discovered in the previous step. Thus, the MFS mined by MSPS provides a border under which all frequent sequences exist.

The main contributions of this research are:

(1) A new MSPS algorithm is developed for mining maximal frequent sequences. It uses a sampling technique to look ahead long patterns and then performs a bottom-up levelwise mining. MSPS outperforms GSP considerably, and it also shows better scalability than SPAM [4] and SPADE [23].
(2) How to trade-off the cost and the quality of sampling is studied thoroughly. For association rule mining, they proposed to lower the user-specified minimum support (denoted by *minsup*) for the mining of a sample to reduce the number of misses (i.e., false negatives). They usually calculate the minsup for the sample by using the Chernoff boundary without assuming any distribution for the support of a sequence in the sample. As a result, the calculated minsup for the sample could be too low, such that it produces many overestimates (i.e., false positives) [18]. For the case of sequence mining, where the search space is much larger, lowering a small minsup for the sample often makes the cost of mining the sample and verifying the result too high due to many overestimates. However, we used the normal distribution to model the support of a sequence in the sample, and analyzed how to increase the user-specified minsup to mine the sample for the best overall performance.
(3) Optimization components are developed to reduce the computation complexity of the maximal frequent sequence mining. A signature-based method and a hash-based method are developed to perform a partial subsequence infrequency-based pruning when the set of frequent sequences at some level is too big to be loaded into memory. A new prefix tree structure and a customer sequence trimming technique are also developed to count the candidate sequences of different sizes efficiently by reducing the number of database scans and the computational cost.

The rest of the paper is organized as follows: Sect. 2 introduces the basic concepts of sequence mining. Section 3 reviews some related works. Section 4 describes the MSPS algorithm, and Sect. 5 theoretically analyzes the sampling in

MSPS. The experimental results and performance analyses are presented in Sect. 6. Section 7 contains some conclusions and future work.

2 Sequence mining

Let $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$ be a set of items. An *k-itemset* i is a set of k items denoted by $\{i_{m_1}, i_{m_2}, \ldots, i_{m_k}\}$, where $1 \leq m_1 < m_2 < \cdots < m_k \leq n$. A sequence s is an ordered list of itemsets denoted by $\langle s_1, s_2, \ldots, s_k \rangle$, where each s_i, $1 \leq i \leq k$, is an *itemset*. A sequence $s_a = \langle a_1, a_2, \ldots, a_p \rangle$ is contained in another sequence $s_b = \langle b_1, b_2, \ldots, b_q \rangle$ if there exist integers $1 \leq j_1 < j_2 < \cdots < j_p \leq q$ such that $a_1 \subseteq b_{j_1}, a_2 \subseteq b_{j_2}, \ldots, a_p \subseteq b_{j_p}$. If s_a is contained in s_b, s_a is a *subsequence* of s_b, and s_b is a *supersequence* of s_a. An item may appear at most once in an itemset, but it may appear multiple times in different itemsets of a sequence. If there are k items in a sequence, the length of the sequence is k, and we call it a k-sequence. For example, a 3-sequence $\langle \{A\}, \{B, C\} \rangle$ is a subsequence of a 5-sequence $\langle \{C\}, \{A, D\}, \{B, C\} \rangle$. For simplicity, these two sequences can be represented as $A - BC$ and $C - AD - BC$.

Given a database $\mathcal{D}$ of customer transactions, each transaction consists of a customer-id, transaction-time, and an itemset that includes all the items purchased by the customer in that single transaction. All the transactions of a customer can be viewed as a customer sequence, where these transactions are ordered by their transaction times. We denote a customer sequence t as $\langle T_1, T_2, \ldots, T_m \rangle$, which means the customer has m transactions in the database and each transaction T_i, $1 \leq i \leq m$, contains all the items purchased in that transaction. A customer supports a sequence if the sequence is contained by the customer sequence. The support for a sequence in database $\mathcal{D}$ is defined as the fraction of total customers who support the sequence. Given a user-specified minimum support, denoted by *minsup*, a sequence is frequent if its support is greater than or equal to minsup. The problem of sequence mining is to find all the frequent sequences in the database with respect to a user-specified minsup. If a sequence is frequent and none of its supersequences is frequent, then it is a maximal frequent sequence.

Based on the above definitions, two properties are often utilized to speed up the sequence mining: (1) Any supersequence of an infrequent sequence is not frequent, so it can be pruned from the set of candidates. This is called subsequence infrequency-based pruning. (2) Any subsequence of a frequent sequence is also frequent, so it can be pruned from the set of candidates. This is called supersequence frequency-based pruning.

3 Related work

Mining sequential patterns was introduced in Agrawal and Srikant [3] with AprioriAll, AprioriSome, and DynamicSome algorithms. Although AprioriSome and DynamicSome try to generate and count long candidate sequences before enumerating all their subsequences, their performance is usually worse than that of AprioriAll. The reason is that too many false candidates are generated without being pruned by the subsequence infrequency-based pruning. The performance gain

from the supersequence frequency-based pruning is not enough to offset the cost of counting so many false candidates.

GSP [17] was proposed for generalized sequence mining, and it requires multiple passes on the database. At pass k, the set of candidate k-sequences are counted on the database, and frequent k-sequences are determined. Then, the candidate $(k+1)$-sequences are generated by joining frequent k-sequences for the next pass. This process will continue until no candidate is generated. Even though GSP is much faster than AprioriAll, it has a very high overhead of enumerating every single frequent subsequence when there are some long patterns. This is also the main weakness of other Apriori-like algorithms, such as PSP [12]. For PSP, a prefix tree was developed as the internal data structure to organize and count candidates more efficiently. The differences between the PSP's prefix tree and the one developed for our MSPS will be discussed later.

SPADE [23] works on the databases with a vertical id-list format, where a list of (customer-id, transaction-time) pairs is associated with each sequence, and the candidates are counted by intersecting the id-lists. A lattice-theoretic approach is used to decompose the search space into small pieces so that all working id-lists can be loaded into memory. PrefixSpan [15] projects a large sequence database recursively into a set of small postfix subsequence databases based on the currently mined frequent prefix subsequences. Then, the subsequent mining is confined to each small projected database. A memory-based pseudoprojection technique is developed to save the computation cost of projection and the memory space for projected databases. SPAM [4] uses a vertical bitmap representation of the database for candidate generation and counting. A bitmap is created for each item in the database, where each bit corresponds to a transaction. If transaction j contains item i, then bit j in the bitmap for item i is set to 1; otherwise, it is set to 0. SPAM also uses a depth-first traversal of the Lexicographic sequence tree and an Apriori-based pruning of candidates.

DISC-all [8] uses a new strategy named Direct Sequence Comparison (DISC) to prune the search space. The basic idea is that, when mining k-sequences, it sorts the customer sequences in the database based on their *k-minimum subsequences*, which are k-sequences smaller than any other k-sequences in terms of alphabetical order and the transaction id of their items. As the customer sequences containing the same k-minimum subsequences are listed consecutively after sorting, we can easily check if the first k-minimum subsequence is frequent or not. Then, from the customer sequences containing this first k-minimum subsequence, we can identify a group of conditional k-minimum subsequences and update the sorted database based on them. This procedure is repeated until all the frequent k-sequences are found. In this way, many infrequent candidate k-sequences which are not appearing in the database can be skipped without being counted. Furthermore, DISC-all also combines the subsequence infrequency-based pruning, database projection, and customer sequence trimming to improve its performance.

SPADE, PrefixSpan, SPAM, and DISC-all were reported more efficient than GSP. However, their performance may not be scalable in certain cases. For SPADE, if the database is in the horizontal format, where the transactions form the tuples in the database, transforming it to the vertical format requires extra disk space that is almost the same as its size. This may be a problem in practice if the database is large. Even if the database is in the vertical format, to efficiently

count 2-sequences, SPADE proposes transforming it back to the horizontal format on the fly. This usually requires much time and memory for very large databases and results in a performance degradation, which is shown in our tests. SPAM is more efficient in mining long patterns than other algorithms. However, it consumes more memory space than SPADE and PrefixSpan. It is claimed to be a memory-based algorithm. According to our tests, its scalability is much more sensitive to the number of items and the database size than other algorithms. The comparison between MSPS, GSP, SPADE, and SPAM is presented in detail in the performance analysis section.

Even though we could not test PrefixSpan and DISC-all because we did not have the source codes, we can still estimate their scalability theoretically. PrefixSpan may be challenged when the database has a large number of customer sequences and items. A large number of items often produce many combinations at the early stage of mining with a small minimum support level, and it requires PrefixSpan to construct more projected databases. If the database is very large, the cost of projection will be high and much more memory is necessary. In addition, PrefixSpan performs a depth-first search. Thus, the largest number of projected databases to be resident in memory at the same time is same as the length of the longest frequent sequence. When the minimum support is small, even with the pseudoprojection of the database, the memory requirement can be easily much more than the available memory space, because the size of the projected databases based on the frequent 1-sequences alone can be almost the same as the original database size without including the amount of memory required for the pseudoprojection at the lower levels during the depth-first search.

A major problem of DISC-all is the sorting of the customer sequences based on their k-minimum subsequences. When the database is large, for example, with millions of customer sequences, the sorting could be very time-consuming, because the comparison between customer sequences with certain length is not trivial and we may not have enough memory to perform a quick memory-based sorting. Moreover, DISC-all needs to update the sorted database many times to mine all the frequent sequences with the same length. For a large database with many items, say 10,000 items, the number of distinct k-sequences appearing in the database is usually very big, and thus results in a very high cost for the re-sorting. The test results reported in Chiu et al. [8] could not show the scalability of DISC-all even though the biggest database used is just about 50 Mbytes and contains only 1000 items, which is about 10% of the biggest database used in our research (500 Mbytes and 10,000 items).

There are not many algorithms proposed for mining closed frequent sequences due to its complexity. CloSpan [20] performs a depth-first search on the Lexicographic sequence tree. At each node, CloSpan applies a pruning technique called Early Termination by Equivalence to see if the subtree rooted at the current node can be absorbed by any other potential closed frequent sequence already found. Finally, a candidate set of closed frequent sequences is determined, and a postpruning step is needed to exclude nonclosed frequent sequences. As discussed in Wang and Han [19], CloSpan follows the candidate maintenance-and-test paradigm, which requires much memory to maintain all the historical candidate closed frequent sequences to do the closure checking for the newly found closed sequence. Thus, the algorithms with this paradigm have rather poor scala-

bility when the minimum support is low or patterns are long. The BIDE algorithm [19] can solve this problem by adopting a novel sequence closure checking scheme named Bidirectional Extension. With this, BIDE does not need to store any historical closed patterns. BIDE also performs a depth-first search on the Lexicographic sequence tree. At each node, the forward directional extension is used to grow the frequent prefix sequences and check the closure of the prefix sequences, while the backward directional extension is used for both closure checking and search space pruning. However, unlike other algorithms, the current version of BIDE can mine only the closed frequent sequences of single items, instead of the sequences of itemsets. It is noticed that, like PrefixSpan, both CloSpan and BIDE perform the pseudoprojection of the database along the path of the depth-first search. This poses the same scalability problem as in PrefixSpan when the minimum support is low because we must have enough memory to hold almost the whole database and associated pseudoprojection data structures.

In previous researches on sampling [7, 10, 11, 18, 21, 22], the focus was on two aspects: (1) how to choose the sample size, and (2) how to avoid missing patterns in the sample. In Chen et al. [7], the FAST algorithm progressively refines the initial sample to obtain a small final sample and reports the set of frequent itemsets in the final sample as the result. In Toivonen [18] and Zaki et al. [22], the Chernoff boundary was used to choose the sample size and to lower the user-specified minsup to mine the sample.

A new probabilistic framework was developed in Yang et al. [21] to mine the sequences in a noisy environment. However, the proposed method defines a probabilistic match only between the sequences of items, not between the sequences of itemsets. For example, the match between $A - B$ and AB is not defined. Thus, it cannot mine the frequent sequences of itemsets. The Chernoff boundary was also used in Yang et al. [21] to estimate if a sequence is frequent or not. The sequences whose match is very close to the user-specified minimum match are considered as ambiguous patterns, and they have to be verified against the whole database. They theoretically proved that if a sequence is frequent, then its probability of being missed in the sample is small. However, to reduce the number of misses in the sample, they also suggested to lower the minimum match for the sample as in Toivonen [18] and Zaki et al. [22].

In Domingo et al. [10, 11], they proposed how to dynamically determine the sample size based on the estimated support of the candidate itemsets, which is expected to be tighter than the sample size based on the Chernoff boundary. The basic idea is that a relatively small sample can be used if the support of a candidate itemset is far from the minimum support. The transactions are randomly selected from the database and added into the sample one by one. Each candidate itemset is evaluated based on the current sample to see if it needs to be evaluated with more transactions at the next step. However, they tested this method with only one candidate itemset. Thus, as suggested in Domingo et al. [10, 11], more work needs to be done to combine this adaptive sampling method with some mining algorithms, like Apriori, and more experiments are required to test the cost of the online sampling.

Unlike the previous sampling methods, we used the normal distribution to model the support of a sequence in the sample, instead of using the Chernoff boundary. Moreover, instead of trying to reduce the misses in the sample ei-

ther by increasing the sample size too much or by lowering the user-specified minsup support to mine the sample, we proposed to increase the small user-specified minsup a little bit to mine the sample, so that we can achieve the best overall performance.

4 MSPS algorithm

Like GSP, MSPS also uses the candidate generation then counting approach to perform the mining, but the performance is improved very much by combining the supersequence frequency-based pruning into its bottom-up, breadth-first search. It has the following original components: (1) A signature-based approach and a hash-based approach are developed to perform a partial subsequence infrequency-based pruning when the set of frequent k-sequences is too big to be loaded into memory totally for the generation of candidate $(k+1)$-sequences. (2) To efficiently count candidates of different sizes, a prefix tree structure is developed, and it also facilitates the customer sequence trimming. (3) To support supersequence frequency-based pruning, sampling is used to find long frequent patterns early. (4) To make the sampling more efficient and robust in sequence mining, a theoretical method of adjusting the small user-specified minsup for mining the sample database is proposed.

In this section, we first present an overview of MSPS and then describe the candidate generation and pruning, candidate counting, and sampling in detail. The following notations will be used in our description: DB is the original database and db is a small random sample of DB. If a sequence is frequent in DB, it is called a global frequent sequence. L_k^{DB} is the set of all global frequent k-sequences and C_k^{DB} is the set of all candidate k-sequences generated from L_{k-1}^{DB}. MFS^{DB} is the set of all global maximal frequent sequences in the whole database. If a sequence is frequent in the sample db, we call it a local frequent sequence. L_k^{db}, C_k^{db}, and MFS^{db} are the set of local frequent k-sequences, the set of candidate k-sequences generated from L_{k-1}^{db}, and the set of local maximal frequent sequences in the sample, respectively. $LongFS^{DB}$ is the set of verified long global frequent sequences found from the sample result.

4.1 Description of MSPS

The basic idea of MSPS is simple: if some long frequent patterns are found early, they can be used to prune the search space so that the mining can speed up. To find long frequent patterns, a small sample db is mined first. We must balance the gain from the supersequence frequency-based pruning and the cost for mining the sample and then verifying the sample result. MSPS consists of three phases:

Phase 1: L_1^{DB} and L_2^{DB} are determined. Candidate 3-sequences are generated from L_2^{DB}. To count candidate 2-sequences, a matrix is used. The entry at position (i, j) in the upper-triangle of the matrix contains the counts of three candidates: $i-j$, ij, and $j-i$.

Phase 2: A random sample is drawn from *DB*, then how much the user-specified minsup should be adjusted for mining the sample is determined. We mine the sample starting with C_3^{DB} in a bottom-up, breadth-first manner. The local maximal frequent sequences are extracted to construct MFS^{db}. Then, we perform a top-down search to find long global frequent sequences by using MFS^{db}. All those sequences in MFS^{db} are considered as global candidates and counted against *DB*. If a k-sequence, $k > 3$, is infrequent, all of its $(k-1)$-subsequences are considered as candidates for the next pass. For a frequent k-sequence, we stop splitting it, and put it into the set $LongFS^{DB}$ if none of its supersequences is already in this set. For a newly generated candidate $(k-1)$-sequence, if it has any supersequence in $LongFS^{DB}$, we remove it from further consideration. We also check if the newly generated candidate $(k-1)$-sequence has any subsequence which is already identified as infrequent. If yes, this candidate $(k-1)$-sequence must be split again. At the end of this Phase 2, $LongFS^{DB}$ contains all the verified long frequent sequences found by using the sample result.

Phase 3: The bottom-up search suspended at the end of Phase 1 is resumed from pass 3. With $LongFS^{DB}$, we can apply the supersequence frequency-based pruning on the candidates generated at each pass. The candidates which appear in $LongFS^{DB}$ or have any supersequence in $LongFS^{DB}$ do not need to be counted. They are simply considered as frequent and used for the candidate generation for the next pass. Finally, MFS^{DB} is extracted from all global frequent sequences found.

While the sample result helps us identify most long frequent sequences quickly, the bottom-up mining of MSPS still generates all the frequent sequences. Some of them are actually counted against the whole database, and others are not counted as they are pruned by using the long frequent sequences found from the sample. Thus, we will not miss any real maximal frequent sequences in our final mining result.

Figure 1 shows how MSPS mines a database *DB* of eight customer sequences. A random sample db is 50% of *DB*. The user-specified minsup is 50%, and it is also used to mine the sample. In Phase 2, we mine db starting with C_3^{DB} and finally obtain MFS^{db}. Then, a top-down search is performed to verify the patterns in MFS^{db}. In this example, all the three patterns in MFS^{db} are globally frequent, so they are put into $LongFS^{DB}$. In Phase 3, at each pass, we use $LongFS^{DB}$ to prune the candidates. In each C_k^{DB}, the candidate sequences in grey color are the ones pruned, and only the ones in black color are actually counted against *DB*. Thus, with sampling and supersequence frequency-based pruning, MSPS largely reduces the number of candidates to be counted against *DB*. To mine this database, GSP needs to count 26 candidates against *DB* from pass 3, but MSPS counts only three candidate sequences.

4.2 Candidate generation and pruning

In both Phases 2 and 3, we have performed the bottom-up, breadth-first search on the sample and the original database, respectively. At pass k, the candidates are generated in two steps:

DB	ABC-DE, ABC-E-DE, ABC-F-DE, ABC-DF-A-K, B-ABC-DE, ABC-B-DEF, BC-ABC-DE, ABC-BCD-DEF
Random Sample db (50% of DB)	ABC-E-DE, ABC-F-DE, ABC-B-DEF, ABC-BCD-DEF

Phase 1

$\|F_1^{DB}\| = 6$	$\|F_2^{DB}\| = 14$	$\|C_3^{DB}\| = 18$
A, B, C, D, E, F,	AB, AC, A-D, A-E, A-F, BC, B-B, B-D, B-E, B-F, C-D, C-E, C-F, DE	ABC, AB-D, AB-E, AB-F, AC-D, AC-E, AC-F, A-DE, BC-D, BC-E, BC-F, B-BC, B-B-B, B-B-D, B-B-E, B-B-F, B-DE, C-DE

Phase 2 *

$\|F_3^{db}\| = 16$	ABC, AB-D, AB-E, AB-F, AC-D, AC-E, AC-F, A-DE, BC-D, BC-E, BC-F, B-B-D, B-B-E, B-B-F, B-DE, C-DE
$\|C_4^{db}\| = 7$	ABC-D, ABC-E, ABC-F, AB-DE, AC-DE, BC-DE, B-B-DE
$\|F_4^{db}\| = 7$	ABC-D, ABC-E, ***ABC-F***, AB-DE, AC-DE, BC-DE, ***B-B-DE***
$\|C_5^{db}\| = 1$	ABC-DE
$\|F_5^{db}\| = 1$	***ABC-DE***
$\|MFS^{db}\| = 3$	***B-B-DE, ABC-F, ABC-DE***
$\|LongFS^{DB}\| = 3$	B-B-DE, ABC-F, ABC-DE

Phase 3

$\|C_3^{DB}\| = 18$ (3 left after pruning)	ABC, AB-D, AB-E, AB-F, AC-D, AC-E, AC-F, A-DE, BC-D, BC-E, BC-F, B-BC, B-B-B, B-B-D, B-B-E, B-B-F, B-DE, C-DE
$\|F_3^{DB}\| = 15$	ABC, AB-D, AB-E, AB-F, AC-D, AC-E, AC-F, A-DE, BC-D, BC-E, BC-F, B-B-D, B-B-E, B-DE, C-DE
$\|C_4^{DB}\| = 7$ (0 left after pruning)	ABC-D, ABC-E, ABC-F, AB-DE, AC-DE, BC-DE, B-B-DE
$\|F_4^{DB}\| = 7$	ABC-D, ABC-E, ***ABC-F***, AB-DE, AC-DE, BC-DE, ***B-B-DE***
$\|C_5^{DB}\| = 1$ (0 left after pruning)	ABC-DE
$\|F_5^{DB}\| = 1$	***ABC-DE***
$MFS^{DB} = 3$	***B-B-DE, ABC-F, ABC-DE***

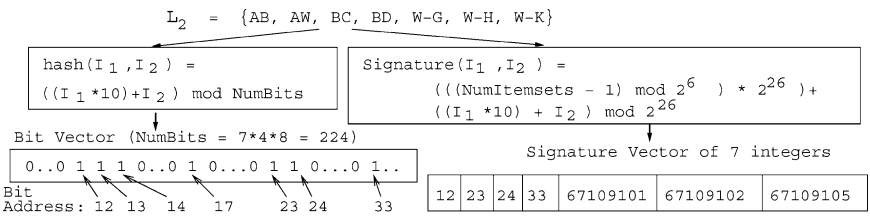
* In phase 2 of MSPS, F_k^{db} and MFS^{db}only include the patterns which can be grown from C_3^{DB}.

Fig. 1 An example of MSPS

Join Step: we generate local (global) candidate $(k+1)$-sequences by joining L_k^{db} with L_k^{db} (L_k^{DB} with L_k^{DB}) as in the GSP algorithm. For any two local (global) frequent k-sequences, say s_1 and s_2, in L_k^{db} (L_k^{DB}), if the subsequence obtained by dropping the first item of s_1 is the same as the subsequence obtained by dropping the last item of s_2, a new candidate is generated by extending s_1 with the last item of s_2. The added item starts a new itemset in s_1 if it was a separate itemset in s_2; otherwise, it becomes a member of the last itemset in s_1.

Prune Step: In both Phases 2 and 3, the subsequence infrequency-based pruning is applied. The local (global) candidate $(k+1)$-sequences with any subsequence of length k which is not in L_k^{db} (L_k^{DB}) are removed. Especially in Phase 3, since we have $LongFS^{DB}$, the supersequence frequency-based pruning also can be performed. Thus, we remove global candidate $(k+1)$-sequences which are in $LongFS^{DB}$ or have any supersequence in it.

A weakness of GSP is the way that a large L_k^{DB} is processed. When the user-specified minsup is very small, L_k^{DB} could be too large to be loaded into memory totally. For this case, GSP proposed to use a relational merge-join technique



C_3	Apriori Pruning	Hash-based Pruning	Signature-based Pruning
ABC	Yes	No	Yes
ABD	Yes	No	Yes
AW-G	Yes	No	Yes
AW-H	Yes	Yes	Yes
AW-K	Yes	Yes	Yes

Fig. 2 Comparison of hash-based and signature-based pruning

to generate candidates. But in this manner, the subsequence infrequency-based pruning cannot be applied because the whole L_k^{DB} is not available in memory and retrieving the relevant portions of L_k^{DB} from a disk requires too many swaps. Without the subsequence infrequency-based pruning, usually the performance of GSP degrades a lot.

For MSPS, we have tried two methods to deal with this problem: hash-based pruning and signature-based pruning. Figure 2 shows an example to illustrate the two pruning methods, and it also demonstrates that the signature-based pruning is more effective when the same amount of memory is used. For each frequent sequence, we can calculate its hash value and signature by using a hash function and a signature function, respectively. In the hash-based pruning, we use a bit vector. If there is at least one frequent sequence hashed to a bit address in the bit vector, the corresponding bit is "1"; otherwise, the bit is "0." In the signature-based pruning, a signature vector is used to contain all the unique signatures of the frequent sequences. These signatures are sorted in ascending order in the signature vector. Compared with the case of loading all the frequent sequences into memory, the bit vector and the signature vector take much less memory, especially when the frequent sequences are long. Both vectors can be loaded into memory totally.

In this example, suppose that L_2 contains 7 frequent 2-sequences. For a fair comparison, we used the same memory space for the bit vector and the signature vector. For the signature vector, we need 7 integers of 4 bytes for all the unique signatures. So, the bit vector can have $7 \times 4 \times 8 = 224$ bits to use the same amount of memory. The typical hash function used in this example was also used in Park et al. [14] and Shintani and Kitsuregawa [16], and I_1 and I_2 denote the lexicographic order of the items. In the hash function, we use the modulo operation to map the value of $I_1 \times 10 + I_2$ into the range between 0 and 223, because the bit vector has only 224 bits. For example, the hash value of AW is $(1 \times 10 + 23)\ mod\ 224 = 33$, so that the bit vector has "1" at its bit address 33.

The given signature function is little more complex, where *NumItemsets* represents the number of itemsets in the frequent sequence. For example, the signature of $W - G$ is $((2-1)\ mod\ 2^6)2^{26} + (23 \times 10 + 7)\ mod\ 2^{26} = 67109101$. It is the fifth element in the signature vector. We will explain the design idea of this signature function later.

We can generate C_3 by joining L_2 with itself as described in Sect. 4.2, and they are shown in Fig. 2. In the case that L_2 can be totally loaded into memory, we are able to perform the subsequence infrequency-based pruning used in Apriori, and all candidate 3-sequences can be pruned. With the hash-based pruning, we cannot prune the first three candidates due to the collision in hashing. For example, the candidate ABC is pruned by the Apriori pruning because its subsequence AC is not in L_2. But in the hash-based pruning, ABC is not pruned because the hash value of AC is 13 and the corresponding bit in the bit vector is "1," which was actually set by a frequent 2-sequence $W - G$. However, in the signature-based pruning, the signature of AC, which is 13, is not in the signature vector. Thus, we can detect that AC is infrequent and then remove ABC.

In our example, the 32-bit integer signature has two parts: (1) the lowest 26 bits encode the items and their order in the sequence, and (2) the highest 6 bits encode the number of itemsets in the sequence. The first part of the signature is determined by $(I_1 \times 10 + I_2)\ mod\ 2^{26}$ in the signature function, and it is similar to the hash function used in our example. But, the difference is that the value of $I_1 \times 10 + I_2$ is mapped into a much bigger range $[0, 2^{26} - 1]$, which largely reduces the chance of collision between signatures. The second part of the signature is determined by $((NumItemsets - 1)\ mod\ 2^6)2^{26}$ in the signature function, and it can distinguish the sequences like BC and $B - C$. Thus, the information of a sequence is better represented by a signature than a hash value, and the signature-based pruning is usually more effective than the hash-based pruning when the same amount of memory is used.

Note that how to segment the 32-bit signature to encode different information of a sequence has nothing to do with the signature vector size. In our example, as we expect the length of the longest frequent sequence would be less than 64, the second part of the signature takes only 6 bits, and the remaining 26 bits are used for the first part.

The general forms of the hash function and the signature function of a sequence can be expressed as follows:

$$hash\ value = \left(\sum_{i=1}^{k}(I_i \times 10^{k-i})\right)\ mod\ NumBits$$

$$\begin{aligned} signature = {} & (((NumItemsets - 1)\ mod\ 2^p)2^{q+r}) \\ & + \left(\left(\left(\sum_{j=1}^{NumItemsets}(NumItems_j \times 10^{NumItemsets-j})\right)\ mod\ 2^q\right)2^r\right) \\ & + \left(\left(\sum_{i=1}^{k}(I_i \times 10^{k-i})\right)\right)\ mod\ 2^r \end{aligned}$$

Here, k is the total number of items in the sequence; I_i denotes the ith item in the sequence; $NumItemsets$ is the number of itemsets in the sequence; and $NumItems_j$ is the number of items in the jth itemset in the sequence.

For the signature-based pruning, we segment the 32-bit signature into three parts. The lowest r bits encode the items and their order in the sequence, the middle q bits encode the information about the number of items in each itemset in the

sequence, and the highest p bits encode the number of itemsets in the sequence. This general signature function is a little bit different from the one used for the example in Fig. 2. By encoding the information about the number of items in each itemset in the sequence, we can distinguish the sequences like $AB - DA$ and $ABD - A$. In practice, we can set the parameters p, q, and r as 5, 7, and 20, respectively, which show very good pruning effectiveness.

Comparing with the hash-based pruning, the signature-based pruning can be more effective when the same memory space is used. Our devised signature function encodes not only the items and their order, but also the number of itemsets and the size of each itemset. By encoding these information into different bits, it actually partition the signatures of the sequences with different number of itemsets and itemset sizes into different value ranges. Such partitioning reduces the chance of collisions. In addition, because the signature value can be as big as the maximum value that a unsigned integer can represent, the signatures can have much bigger value range than hash values, and hence have less probability of collisions.

A practical problem of hash technique is that even though the hash vector can have millions of bits, a lot of them have "0" value. A good hash function can produce randomly distributed hash values, but it is not so easy to get a good randomizing hash function for the databases containing various data distributions and skewness. For the signature-based pruning, if the signature vector has N entries, we can guarantee that each of those entries covers at least one pattern. So, no entry is empty and the memory space allocated is fully utilized. Thus, when the same memory space is used, the signature technique can distinguish more patterns.

A disadvantage of the signature-based pruning is that the computation complexity is $O(\log N)$ due to the binary search in the signature vector when N is the number of entries in the signature vector. However, it is $O(1)$ for hash-based pruning. The extra search cost can be easily compensated as more candidate sequences can be pruned by the signature-based pruning, because counting those false candidates against millions of customer sequences usually costs even more.

With the signature-based subsequence infrequency-based pruning, MSPS performs much better than GSP when the seed set of frequent sequences for generating the candidates cannot be loaded into memory totally. If the memory cannot hold all the candidates generated, they can be processed part by part.

4.3 Counting of candidate sequences

During the top-down search for long patterns covered by MFS^{db}, to reduce the number of passes, we need to count candidates of different sizes at each pass over the database. For that purpose, we developed a new prefix tree structure. Since it is much more efficient than the hash tree, we also use it to count the candidates of the same size during the bottom-up search in Phases 2 and 3. We first describe our prefix tree and the customer sequence trimming technique, and then compare it with the prefix tree used for PSP [12].

4.3.1 Overview of the prefix tree and the customer sequence trimming

The following example shows how the prefix tree works. Suppose we have 10 candidates of length 2 or 3. The prefix tree is constructed as shown in Fig. 3. Each

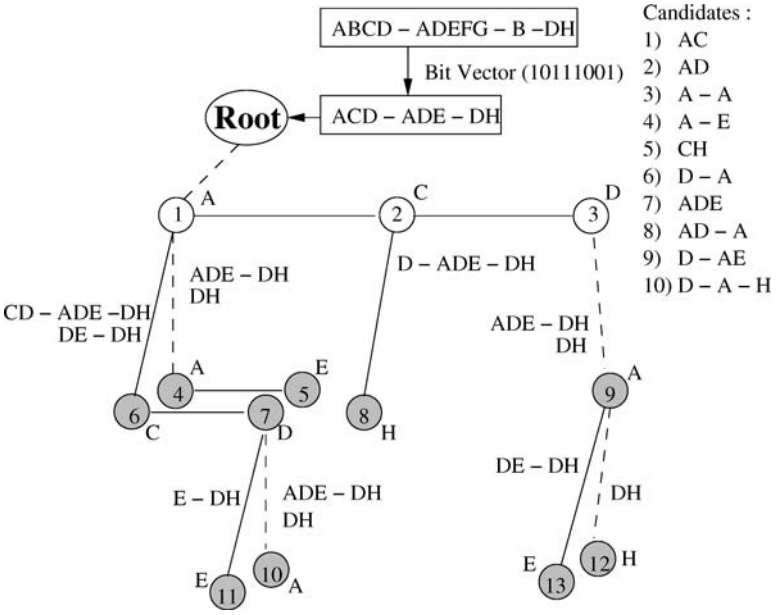


Fig. 3 Prefix tree of MSPS

node is associated with a pointer. If the path from the root to a node represents a candidate, the pointer points to the candidate; otherwise, it is NULL. A node may have two types of children. The "I-extension" child means the item represented by the child node is in the same itemset with the item represented by its parent node. The "S-extension" child means the item represented by the child node starts a new itemset. All the S-extension (I-extension) children of a node are linked together, and only the first child is linked to their parent node by a dashed (solid) line. For example, nodes 4 and 5 are the S-extension children of node 1, and the corresponding paths represent the candidates $A-A$ and $A-E$, respectively. Nodes 6 and 7 are I-extension children, and their paths represent AC and AD, respectively.

To speed up the counting, a bit vector is associated with the prefix tree to facilitate the customer sequence trimming. In this example, we have eight items in the database: A, B, C, D, E, F, and H. Since B, F, and G do not appear in any candidate, they should be ignored during the counting. Thus, the bit vector is set as (10111001), where 1 at the ith bit position means item i appears in the prefix tree. All the bits are initialized to 0, and the corresponding bits are set to 1 as we insert candidates into the prefix tree.

Given a customer sequence $s = ABCD - ADEFG - B - DH$, we first trim it to $s' = ACD - ADE - DH$ using the bit vector. Then, a recursive method is used to count all the candidates contained in s'. At the root node, we check each item in $ACD - ADE - DH$ to see if it is in the root node's S-extension children. The first item of s' is A, and it appears as the first S-extension child of the root node. So we recursively call the count function at the root node with two sequence segments. The segment $CD - ADE - DH$ is used in the call for node 1's I-extension link, while $ADE - DH$ is for its S-extension link. Then, we can locate the second item

of s', C, at node 2. Since node 2 has no S-extension child, only one recursive call with the segment $D - ADE - DH$ is made for its I-extension link. The third item of s', D, is the last item of the first itemset in s'. Only one call with segment $ADE - DH$ is made for node 3's S-extension link. The fourth item of s', A, can be located at node 1 again, and we make two recursive calls. One is for the node 1's I-extension link with $DE - DH$, and the other one is for its S-extension link with DH. Then, we process the remaining items in s', one by one, in the same way. Whenever we locate an item at some node, if the pointer associated with the node is not NULL and the count of the corresponding candidate is not increased yet (for the current customer sequence), it should be increased.

The root node is processed differently from other nodes. At the root node, there is no constraint on which items in the customer sequence should be checked against the root's S-extension link, because the first item of a candidate can appear anywhere in the customer sequence. At other nodes, there are some constraints. Let us see how to make recursive calls at node 1 along its I-extension link. Recall that we have made two recursive calls at the root node with segments, $CD - ADE - DH$ and $DE - DH$, for node 1's I-extension. Now we process them at node 1. Since the two segments are specified for node 1's I-extension link, we just check the items in their first itemsets, CD and DE, against node 1's I-extension link, because only those items are in the same itemset with item A represented by node 1. For $CD - ADE - DH$, since C appears at node 6 which has no child, we stop there by just increasing the count of AC. Another item, D, appears at node 7. We increase the count of AD and make recursive calls for node 7's links. Since D is the last item of the first itemset in $CD - ADE - DH$, only one recursive call with the segment $ADE - DH$ is made for node 7's S-extension link. For another sequence segment $DE - DH$ at node 1, two items of the first itemset, D and E, are checked. D is located at node 7. Since the count of AD is already increased before, we should not increase it again. Two recursive calls are made at node 1 for node 7's links. One is with $E - DH$ for node 7's I-extension link and the other is with DH for the S-extension link. We can ignore E because it is not an I-extension child of node 1. This process will continue until a leaf node is reached or the sequence segment is empty.

4.3.2 Features of the prefix tree and the customer sequence trimming

There are some major differences between our prefix tree and the PSP's prefix tree: (1) Our prefix tree is used to count candidates of different sizes, whereas PSP's prefix tree is only used to count the candidates of the same size. (2) To improve the candidate counting, a bit vector is associated with our prefix tree to facilitate the customer sequence trimming. (3) The supersequence frequency-based pruning reduces the size of our prefix tree when we count the candidates against the whole database.

Due to both (2) and (3), the prefix tree of MSPS is much more efficient. In our prefix tree structure, the I-extension children and S-extension children of a node are linked together, respectively. During the candidate counting, we frequently need to locate the items in the customer sequences along these links. This search could be either sequential or binary depending on how the links are implemented.

Obviously, making the tree smaller or reducing the number of search operations can enhance the counting process. In MSPS, by performing supersequence frequency-based pruning in Phase 3, only a part of the candidate set needs to be processed. Thus, our prefix tree is usually much smaller than PSP's prefix tree at each pass. Moreover, we also reduce the number of search operations by trimming the customer sequences. In PSP, the items not in the prefix tree are not trimmed from the customer sequence. Thus, when these items are processed, they are searched along the corresponding links exhaustively even though they are not in those links. This unnecessary search cost is not trivial when the number of customer sequences is large. MSPS can avoid this problem. As the mining process makes progress, fewer and fewer items would remain in the longer candidate patterns, and the customer sequence trimming can save a lot of time.

Our customer sequence trimming is different from the transaction trimming technique proposed in Park et al. [14]. In Phase 3 of MSPS, those candidates removed by the supersequence frequency-based pruning are not counted, so we do not know exactly how many times an item in a customer sequence appears in the candidate sequences. This prevents us to do the customer sequence trimming as in Park et al. [14]. We can only trim the customer sequences based on the items appearing in the prefix tree.

5 Sampling in MSPS

For both frequent itemset mining and sequence mining, if a pattern is found frequent in db but turns out to be infrequent in DB, it is an *overestimate*. However, if a pattern is infrequent in db but actually frequent in DB, it is a *miss*.

Both our research and Toivonen [18] try to mine the exact result with the help of sampling. While we focused on how to maximize the performance improvement, more attention was given in Toivonen [18] on how to reduce the probability of misses. To achieve that goal, two methods were suggested in Toivonen [18]: (1) mine a large sample, and (2) lower the user-specified minsup for mining the sample. These two methods can reduce the misses but also potentially degrade the overall performance. Mining a large sample cuts the merit of sampling, while lowering the user-specified minsup may generate a large number of overestimates. Obviously, a complete sample result without misses does not necessarily mean the best overall performance. In MSPS, the cost related to sampling includes all the overhead of mining the sample and verifying the sample result, whereas the performance gain is from the supersequence frequency-based pruning. The effectiveness of this pruning is determined by how many long frequent patterns can be found from the sample. As different settings of sample size and the adjusted minsup for mining the sample are used, the overall performance varies accordingly. Thus, we pay our attention to the sample size and the adjusted minsup in the following discussion.

5.1 Sample size

In Toivonen [18] and Zaki et al. [22], the minimum sample size that guarantees a small chance of misses with certain confidence is given by the Chernoff boundary.

Unfortunately, this theoretical guideline is not quite practical because it is too conservative. In MSPS, a large sample can improve the quality of sample result with fewer misses and overestimates. Consequently, verifying the sample result can be done quickly and the supersequence frequency-based pruning can be very effective. But the overhead of mining a large sample is high. However, with a small sample, the overhead of mining sample is low, but MFS^{db} may be in bad quality. Then, the cost of verifying the sample result containing many overestimates would be high. If the small sample size makes the minimum support count for mining the sample (i.e., $minsup \times |db|$ or $lowered\_minsup \times |db|$) very small, mining the sample itself may take a long time. Thus, a small sample does not necessarily mean a lower cost. Furthermore, if only few long frequent sequences are found under the border formed by MFS^{db}, then the supersequence frequency-based pruning will not be effective, either. That is why the sample should not be too large or too small.

In general, we do not know the data distribution characteristic of the database to be mined, so it is hard to determine the best sample size. MSPS allows users to choose a plausible sample size empirically. In our experiments, we set the sample size as 10% of the original database size. By using a default sample size, how to balance the cost related to the sampling and the quality of sample result mainly depends on the adjusted minsup for mining the sample. Even though the default sample size may not be the best one all the time, with the method of adjusting the minsup, it works very well in practice according to our extensive experiments. In Sect. 6.3, we will show the effect of different sample sizes.

5.2 Adjusting the user-specified minimum support for mining the sample

In the sample mining result, a certain rate of misses is tolerable. Our tests show that, for a missing k-sequence, if most of its long subsequences, such as subsequences with length $k-1$ or $k-2$, are found, then the supersequence frequency-based pruning is not affected much. In practice, as long as the sample size is not too small, the probability that most of these subsequences are also missed is quite low. Compared to misses, overestimates could be a bigger problem. Once an infrequent k-sequence is identified frequent in db at pass k, then it may be joined with many other k-sequences to generate a large number of false candidates in mining the sample. Most importantly, the situation may become even worse when the minimum support count for mining the sample is very small. We found this is more serious for sequence mining than for frequent itemset mining, because the search space is much larger. For MSPS, it not only degrades the efficiency of mining the sample, but also causes a high cost to identify the overestimates.

In Toivonen [18], they proposed using the lowered minsup for the sample, however they did not consider the case that the user-specified minsup is very small. In that case, it is dangerous to lower the minsup further. In this research, we investigated how to avoid the overestimates in the case of small user-specified minsup, because such mining task is more time-consuming.

There are three different cases that can happen when MSPS is used: (1) If the user-specified minsup is big, simply using it or even a lowered one to mine the sample works fine. Only a small number of misses and overestimates occur in our tests. This is usually safe because our default sample size is not very small.

(2) If the user-specified minsup is small, the sampling technique is challenged. Using a lowered minsup or even the original user-specified minsup for mining the sample often causes many overestimates because $lowered\_minsup \times |db|$ or $minsup \times |db|$ is too small. Even though increasing the sample size could be a solution for this case, it limits the merit of sampling. Thus, we consider increasing the minsup a little to mine the sample, hoping it will limit the overestimates to a reasonable level. In that case, more misses may occur. However, even though there is a missing pattern, as long as most of its long subsequences are still contained in the sample result, the supersequence frequency-based pruning is not affected much. (3) In some rare cases, the user-specified minsup is extremely small. Then, just increasing the minsup for mining the sample cannot solve the problem. We must consider increasing the sample size too. Actually, both cases (2) and (3) raise the same technical question: when user-specified minsup is small, how to increase the minsup for mining the sample of a certain size? We must keep in mind that if the increase in the minsup for mining the sample is not enough, the problem of overestimates cannot be solved. However, if it is increased too much, we may not find any long patterns from the sample.

For an arbitrary sequence X in a database DB, we have the following theorem.

Theorem 1 *For an arbitrary sequence X in a database DB, whose global support in DB is $P(X)$, the distribution of its local support $P'(X)$ in a random sample db can be approximated by a normal distribution $N(\mu, \sigma^2)$:*

$$\mu = P(X), \quad \sigma = \sqrt{P(X)(1 - P(X))/|db|}$$

where $|db|$ is the sample size, i.e., the number of customer sequences in the sample.

Proof Because the support of X in DB is $P(X)$, the probability that a customer sequence randomly selected from DB contains X is also $P(X)$. For a random sample db with $|db|$ customer sequences that are independently drawn from DB with replacement, the random variable $T(X)$, which represents the total number of customer sequences containing X in db, has a binomial distribution of $|db|$ trials with the probability of success $P(X)$. In general, if $|db|$ is greater than 30, the distribution of $T(X)$ can be approximated by a normal distribution whose mean is $|db| \times P(X)$ and the standard deviation is $\sqrt{|db| \times P(X)(1 - P(X))}$. In MSPS, suppose that we draw a random sample db from DB and then use the point estimator $P'(X) = T(X)/|db|$ to estimate the support of X in the population of DB. Then, $P'(X)$ is also an unbiased estimator with mean $|db| \times P(X)/|db| = P(X)$ and standard deviation $\sqrt{|db| \times P(X)(1 - P(X))/|db|} = \sqrt{P(X)(1 - P(X))/|db|}$. □

For MSPS, it is very important to avoid overestimates from sampling when the user-specified minimum support is small. Thus, we investigated how much the minimum support should be increased to mine the sample of a certain size.

Theorem 2 *In the mining of a database DB for the user-specified minimum support ($minsup \leq 50\%$), if the adjusted minimum support used to mine the sample db is S ($S > minsup$), the probability that an infrequent sequence Y can be overestimated as frequent in db is lower than $1 - P_Z$, where P_Z is the probability of the z-score $Z = (S - minsup)/\sqrt{minsup \times (1 - minsup)/|db|}$.*

Proof By Theorem 1, for an arbitrary sequence X, its local support observed from a sample, $P'(X)$, has a normal distribution with the mean equal to its global support $P(X)$, and the standard deviation is $\sqrt{P(X)(1-P(X))/|db|}$. Since $P(X)(1-P(X)) = -(P(X)-1/2)^2+1/4$ is increasing in the $P(X)$ interval of $[0, 1/2]$, the standard deviation of $P'(X)$ is also increasing in this interval.

Suppose there is a sequence Y_1 and its global support is equal to the user-specified minimum support, i.e., $P(Y_1) = minsup$. Based on the normal distribution, if the adjusted minimum support used for mining the sample is set to S with $S > P(Y_1)$, i.e., $S > minsup$, the probability that Y_1 can be found as a local frequent sequence in db is $1 - P_Z$, where the z-score $Z = (S - P(Y_1))/\sqrt{P(Y_1)(1-P(Y_1))/|db|}$, i.e., $Z = (S - minsup)/\sqrt{minsup(1-minsup)/|db|}$.

Let us consider an infrequent sequence Y_2. By Theorem 1, its local support in db, $P'(Y_2)$, also has a normal distribution with mean $P(Y_2)$ and standard deviation $\sqrt{P(Y_2)(1-P(Y_2))/|db|}$. As discussed earlier, the standard deviation of $P'(Y_2)$ is also increasing in the $P(Y_2)$ interval of $[0, 1/2]$. Since $minsup$ is usually smaller than 50%, in our analysis both $P(Y_1)$ and $P(Y_2)$ are considered to be in that range. As $P(Y_2) < P(Y_1)$, both the mean and the standard deviation of $P'(Y_2)$ should be smaller than those of $P'(Y_1)$, respectively. Therefore, compared with the distribution curve of $P'(Y_1)$, the distribution curve of $P'(Y_2)$ is shifted left and sharper. That means, if we set the adjusted minimum support for mining the sample as S, the probability that an infrequent sequence Y_2 is overestimated as frequent should be lower than $1 - P_Z$, which is the probability that Y_1 can be found as a local frequent sequence in db. □

Finally, from $Z = (S - minsup)/\sqrt{minsup(1-minsup)/|db|}$, we can derive

$$S = minsup + Z \times \sqrt{minsup(1-minsup)/|db|}$$

Theorem 2 has shown the probability of an overestimate in the sample is lower than $1 - P_Z$. In our experiments, the critical value of Z is set to 1.28, where $P_Z = 0.90$, such that the probability of the overestimate is at most 10%. The above formula of S provides a theoretical guideline for adjusting the user-specified minimum support to mine the sample. Even though this adjusted minimum support value may not be the best one all the time, it worked well in most of our experiments. In Sect. 6.3, we will show the effect of different adjusted minimum support values used for mining the sample.

6 Performance analysis

To compare MSPS with other algorithms, we implemented GSP and obtained the source codes of SPAM and SPADE from their authors' Web sites. All the experiments were performed on a SuSE Linux PC with a 2.6 GHz Pentium processor and 1 Gbytes main memory.

MSPS was compared with others on various databases, and we evaluated the scalability of these algorithms in terms of the number of items and the number of customer sequences. We also investigated how the sample size and the adjusted

Table 1 Parameters used in database generation

D	Number of customers in the database
C	Average number of transactions per customer
T	Average number of items per transaction
S	Average length of maximal potentially frequent sequences
I	Average length of maximal potentially frequent itemsets
N	Number of distinct items in the database
N_S	Number of maximal potentially frequent sequences
N_I	Number of maximal potentially frequent itemsets

minsup for mining the sample affect the performance of MSPS. Since the sampling technique is probabilistic, we ran MSPS 100 times for each test. The average execution time of the 100 runs was reported as the performance result. The default sample size was fixed as 10% of the test database for all experiments. The databases used in our experiments are synthetically generated as in Agrawal and Srikant [3]. The database generation parameters are described in Table 1. For all databases, $N_S = 5000$ and $N_I = 25{,}000$; and the names of the databases reflect other parameter values used to generate them.

6.1 Performance comparison

We ran MSPS, GSP, SPADE, and SPAM on four databases with medium sizes of about 100 Mbytes. The number of items in these databases is 10,000. In our tests, SPAM could not mine these databases, and its run was terminated by the operating system. Our machine is a 32-bit system, but the user address space is limited to 2 Gbytes. In all these tests, SPAM always required more than 2 Gbytes of memory, and hence caused the termination.

As discussed before, when the user-specified minsup is small, simply using it or a lowered one to mine the sample may cost too much due to so many overestimates. In practice, we may not know the data distribution characteristics of the database to be mined. Thus, we conservatively assumed that all user-specified minsups in our tests are small and simply increased them a little bit for mining the sample. The adjusted minsup for each test is computed using the formula derived in Sect. 5.2. The probability that an overestimate occurs is set to 10% at most, i.e., $Z = 1.28$. Some typical adjusted minsups computed using the formula are listed in Table 2.

The test results are shown in Fig. 4. With the optimization components integrated, MSPS performs much better than GSP because it processes fewer candidates in a much more efficient way. The advantage of SPADE is the efficient counting of the candidates by intersecting the id-lists. However, when mining a medium-size database with 400,000 customers, the counting for L_2^{DB} in SPADE is

Table 2 Adjusted minsups for mining the sample ($|db| = 400{,}000 \times 10\% = 40{,}000$)

User-specified Minsup (%)	0.4	0.33	0.3	0.25	0.2	0.18
Adjusted Minsup (%)	0.432	0.359	0.327	0.275	0.223	0.201

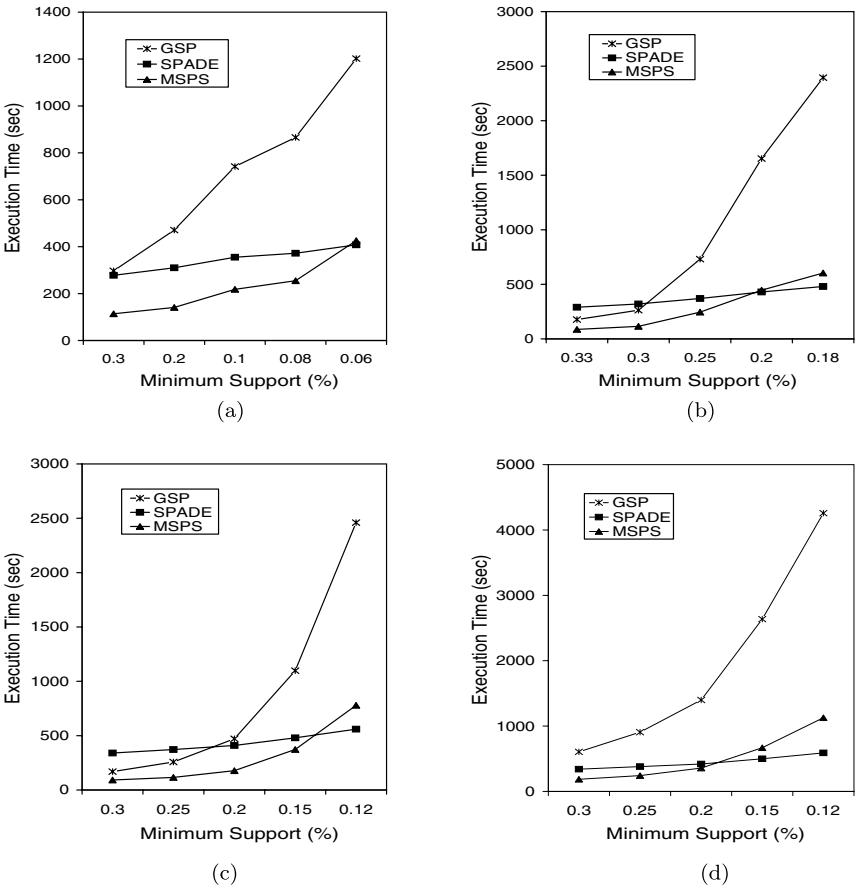


Fig. 4 Performance comparison on medium-size databases. (**a**) D400K-C10-T5-S5-I1.25-N10K. (**b**) D400K-C10-T5-S10-I2.5-N10K. (**c**) D400K-C20-T2.5-S10-I2.5-N10K. (**d**) D400K-C20-T2.5-S10-I1.25-N10K

inefficient and degrades the overall performance very much. Considering both factors, we can say that if there are not enough number of candidates to be counted, SPADE cannot show its efficiency. That is why SPADE is even worse than GSP when the minsup is big, as shown in some of the figures.

When the minsup is decreased, more and more candidates appear during the mining. In that case, the overhead of GSP in candidate generation, pruning, and especially counting using a huge hash tree increases drastically. For MSPS, this situation is considerably improved by using the supersequence frequency-based pruning, the prefix tree structure, and the customer sequence trimming. Figure 5 shows how much the search space can be reduced by MSPS compared to GSP in mining D400K-C20-T2.5-S10-I1.25-N10K. Since counting C_1^{DB} and C_2^{DB} can be done quickly on the database in the horizontal format, we focused on the total number of candidates of length greater than 2, i.e., the candidates from pass 3 in GSP and the candidates in Phase 3 of MSPS. We can see that MSPS can reduce

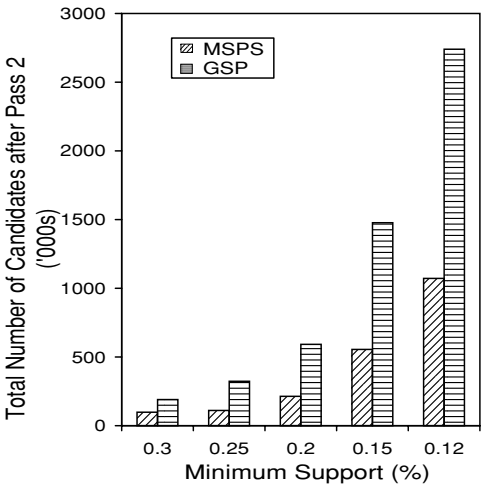


Fig. 5 Search space comparison on D400K-C20-T2.5-S10-I1.25-N10K

the search space by 55–65%. When many passes are required for the mining, most candidates usually appear after pass 2, hence MSPS can outperform GSP further when the minsup is decreased. This improvement also makes MSPS better than SPADE in most tests on the medium-size databases. Only when the minsup is very small, SPADE can beat MSPS.

Figure 6 shows the distribution of the frequent sequences in the databases for the smallest minsup used on them in the tests. Mining D400-C20-T2.5-S10-I1.25-N10K with the minsup of 0.12% produced most frequent sequences, about 2.6 million, while mining D400-C10-T5-S5-I1.25-N10K with the minsup of 0.06% produced only 526K frequent sequences. This difference is reflected in the corresponding execution times of GSP: 4397.5 s versus 1230.5 s. On the contrary, the

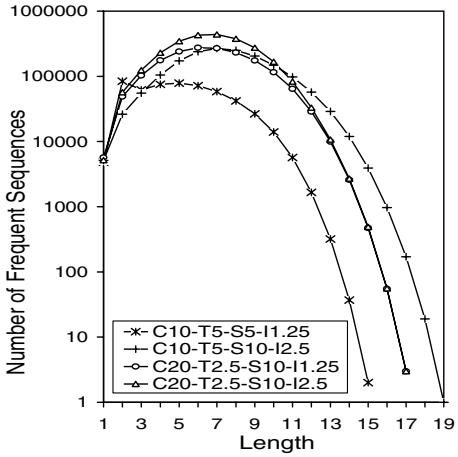


Fig. 6 Distribution of frequent sequences ($D = 400$K, $N = 10$K)

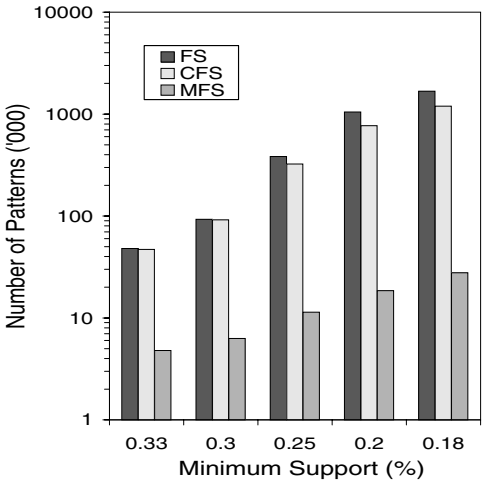


Fig. 7 Comparison of FS, CFS, and MFS

numbers of maximal frequent sequences in these two databases with respect to the corresponding minsup values are 87K and 54K, respectively, and the corresponding execution times of MSPS are 1112.4 and 429.7 s. That means, even if there is a big difference between the numbers of frequent sequences in two databases, the difference between the numbers of maximal frequent sequences is usually much smaller. Thus, compared to frequent sequence mining algorithms, MSPS can be more robust in dealing with the databases with different data distribution characteristics.

Figure 7 shows the numbers of frequent sequences, closed frequent sequences, and maximal frequent sequences in the D400K-C10-T5-S10-I2.5-N10K database with respect to various minsups. When the minsup is decreased from 0.33% to 0.18%, the number of frequent sequences increases very quickly, from 47K to 1.7 million. But the number of maximal frequent sequences increases much slowly, from 4.8K to 28K. That means, mining maximal frequent sequences could be much more scalable. Another interesting observation is that the number of closed frequent sequences is not much smaller than that of frequent sequences for all the cases. For example, when the minsup is 0.18%, the number of closed frequent sequences is 1.2 million, while the number of frequent sequences is 1.7 million. That means, for certain cases, mining closed frequent sequences also might be impractical.

6.2 Scalability evaluation

Both SPADE and SPAM need to store a huge amount of intermediate data to save their computation cost. When the memory space requirement is over the memory size available, CPU utilization drops quickly due to the frequent swapping. Compared with them, MSPS and GSP process the customer sequences one by one, hence only a small memory space is needed to buffer the customer sequences being processed. MSPS can also handle the situation that L_k^{DB} or C_k^{DB} cannot be

totally loaded into memory by using the signatures as explained in Sect. 4. Therefore, MSPS does not require the memory space as much as GSP, SPADE, and SPAM.

Many real-life customer market-basket databases have tens of thousands of items and millions of customers, so we evaluated the scalability of the mining algorithms in these two aspects. First, we started with a very small database D1K-C10-T5-S10-I2.5 and changed the number of items from 500 to 10,000. The user-specified minsup was 0.5%. To run MSPS on such a small database with only 1000 customers, we selected the whole database as the sample and used the user-specified minsup to mine it. When there are only 500 items, the database becomes very dense because each item has a higher probability of being selected by the synthetic database generation program to construct the customer sequences. As a result, all the four algorithms spent much more time to finish the mining. Since MSPS does not apply the sampling on such a small database, supersequence frequency-based pruning is not performed in mining. Thus, in this case, SPADE and SPAM performed better than MSPS and GSP as long as their memory requirement is satisfied.

As the number of items is increased, SPAM shows its scalability problem. Theoretically, the memory space required to store the whole database into bitmaps in SPAM is $D \times C \times N/8$ bytes. For the id-lists in SPADE, it is about $D \times C \times T \times 4$ bytes. But we found these values are usually far less than their peak memory space requirement during the mining, because the amount of intermediate data in both algorithms is quite large. As shown in Fig. 8, even though the D1K-C10-T5-S10-I2.5-N8000 database takes only 260 Kbytes, and the theoretical memory space requirement to store the database in SPAM is about $1000 \times 10 \times 8000/8$ bytes $\approx$ 10 Mbytes, it could not finish the mining when the minsup was 0.5% because it needed more than 2 Gbytes of memory. Compared with SPAM, SPADE divides the search space into small pieces as only the id-lists being processed need to be loaded into memory. Another advantage of SPADE is that the id-lists become

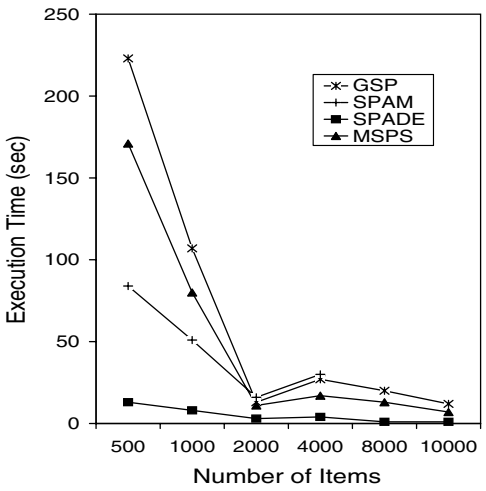


Fig. 8 Scalability: number of items (on D1K-C10-T5-S10-I2.5, minsup = 0.5%)

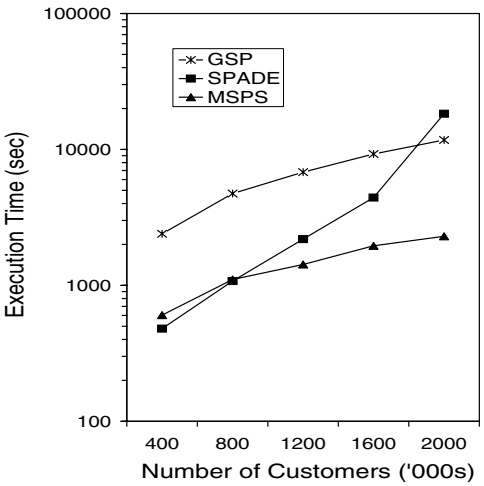


Fig. 9 Scalability: number of customers (on C10-T5-S10-I2.5-N10K, minsup = 0.18%)

shorter and shorter with the progress in mining, whereas the length of the bitmaps does not change in SPAM. These two differences make SPADE much more space-efficient than SPAM.

We also fixed the parameter N as 1000 and changed the database size from 1K to 100K customer sequences. SPAM cannot mine the databases with more than 20K customers due to the memory problem. Our tests showed that SPAM is very sensitive to the number of items and the number of customers, which mainly limits its applicability.

Second, we investigated how they perform on C10-T5-S10-I2.5-N10K when the user-specified minsup is 0.18%. We fixed the number of items as 10,000 and increased the number of customers from 400K to 2000K. SPAM cannot perform the mining due to the memory problem. For SPADE, we partitioned the test database into multiple chunks for better performance when its size was increased. Otherwise, the counting of C_2^{DB} for a large database could be extremely time-consuming. We made each chunk contain 400K customers so that it is only about 100 Mbytes, which is one-tenth of our main memory size. Thus, D400K-C10-T5-S10-I2.5-N10K is processed as one chunk, D800K-C10-T5-S10-I2.5-N10K is divided into two chunks, and so on. Figure 9 shows that the scalability of both MSPS and GSP is quite linear. As the database size is increased, MSPS performs much better than the others.

When the database was relatively small with only 400K customers, SPADE performed best—about 20% faster than MSPS. But SPADE cannot maintain a reasonable scalability as the database becomes larger, and MSPS starts outperforming SPADE. When the database size is increased from 1600K customers to 2000K customers, there is a sharp performance drop in SPADE, such that it is even slower than GSP. In that case, MSPS is faster than SPADE by a factor of about 8. As discussed before, counting C_2^{DB} is a performance bottleneck for SPADE because the transformation of a large database from the vertical format to the horizontal format takes too much time. When the database is very large, the transfor-

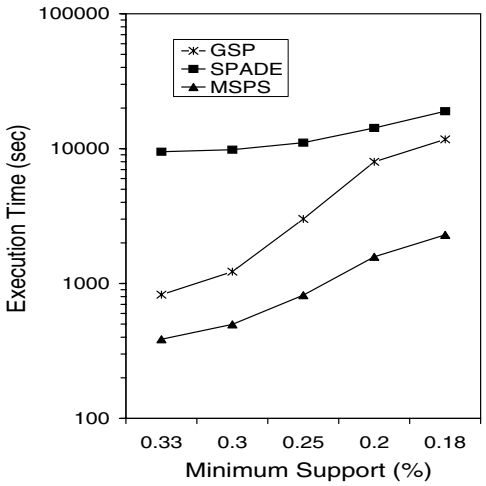


Fig. 10 Performance on a large database D2000K-C10-T5-S10-I2.5-N10K

mation also requires a large amount of memory and frequent swapping, hence the performance drops drastically. Partitioning the database can relieve this problem to some extent but does not solve it completely. In addition, for the database with a large number of items and customers, SPADE needs more time to intersect more and longer id-lists.

Finally, we mined a large database D2000K-C10-T5-S10-I2.5-N10K, which takes about 500 Mbytes, for various minsups. This database was partitioned into five chunks for SPADE, and the results are shown in Fig. 10.

Based on our tests, we found SPADE performs best for small size databases. For medium size databases, MSPS performs better for relatively big minsups, while SPADE is faster for small minsups. When the database is large, SPADE's performance drops drastically, and MSPS outperforms SPADE very much. If the user-specified minsup is big and there are very few long patterns, GSP may perform as well as, or even better than the others due to its simplicity and effective subsequence infrequency-based pruning.

6.3 Effect of the sample size and the adjusted minsup on the performance of MSPS

In MSPS, the overall performance improvement from the supersequence frequency-based pruning is determined by two factors: (1) how much search space can be reduced in Phase 3, and (2) the cost of Phase 2 for mining the sample and verifying the sample result. In this section, we will discuss how the sample size and the adjusted minsup affect the performance of MSPS.

6.3.1 Effect of the sample size on the performance

We ran MSPS on D400K-C10-T5-S10-I2.5-N10K for the minsup of 0.18% with different sample sizes: 5, 10, 20, 40, and 60% of the original database. To observe

Table 3 Effect of the sample size on the performance of MSPS

	Sample size				
	5%	10%	20%	40%	60%
Avg. performance (s)	1148	817	704	755	809
Best performance (s)	630	531	543	708	772
Worst performance (s)	3240	1994	1329	998	933
Std. dev. of performance (s)	492	257	140	42	31
# of runs ($\leq$ 650 s)	1	28	43	0	0
# of runs ($\geq$ 800 s)	81	38	18	7	47
Avg. time of phase 2 (s)	755	481	402	480	536
Avg. time of phase 3 (s)	334	277	244	215	214
Avg. % of search space reduced	75	78	81	82	82

how the sample size alone affects the performance of MSPS, we kept the user-specified minsup unchanged for mining the sample. The experimental results are shown in Table 3.

All the sample sizes we have chosen are not too small to represent the content of the original database to some extent. Even though there may be more misses for a small sample, there is no big difference in how much the search space is reduced in Phase 3. This is because most long subsequences of those missing sequences are still found from the sample. Thus, the supersequence frequency-based pruning is not affected much. However, for a small sample, like 5% of the database, the minimum support count (i.e., $|db| \times minsup$) for mining the sample is very low, hence more overestimates would be produced. The cost of Phase 2 shows that a small sample does not necessarily mean a lower mining cost. The average performance of the 5% sample is not good; and in the worst case, it takes more time than GSP does: 3240 versus 2359 s.

As the sample size is increased, the performance is improved mainly because fewer overestimates are occurred. For the 60% sample, even though the quality of sample result is improved with much fewer misses and overestimates, it requires too much time for mining the sample itself. As a result, we cannot achieve the best performance, either. Compared with others, the 20% sample was the best based on all the measures shown in Table 3: average performance is best (704 s), more than 40% of runs were done in less than 650 s, and only 18 out of 100 runs took over 800 s. It is worth to mention that when we used the 10% sample with the adjusted minsup of 0.201% (instead of the user-specified minsup of 0.18%), we obtained a better average performance of 603 s. It demonstrates that adjusting the minsup (for mining the sample) enables us to use a relatively small sample to achieve a better performance.

6.3.2 Effect of the adjusted minsup on the performance

To evaluate the effect of the adjusted minsup for mining the sample, we used D400K-C10-T5-S10-I2.5-N10K with the user-specified minsup of 0.18%. The sample size was fixed as 10% of the original database. The adjusted minsup values listed in Table 4 were chosen empirically, except for 0.20%, which is based on the formula proposed in Sect. 5.2.

Table 4 Effect of the adjusted minsup on the performance of MSPS

	Adjusted minsup for mining the sample					
	0.17%	0.18%	0.20%	0.22%	0.28%	0.5%
Avg. performance (s)	1331	817	622	691	884	947
Best performance (s)	697	531	528	590	805	941
Worst performance (s)	4682	1994	863	806	926	960
Std. dev. of performance (s)	692	257	66	56	24	3.6
# of runs ($\leq$ 650 s)	0	28	71	30	0	0
# of runs ($\geq$ 800 s)	95	38	2	1	100	100
Avg. Time of Phase 2 (s)	1006	481	160	78	29	11
Avg. Time of Phase 3 (s)	267	277	403	554	796	876
Avg. % of search space reduced	86	78	59	39	10	0.2

As shown in Table 4, when we adjusted the user-specified minsup too big or too small, the overall performance was degraded. Simply using the small user-specified minsup of 0.18% to mine the sample was not the best choice, either. The best adjusted minsup was 0.20%, which is very close to the value computed using our formula (0.201%): 71 out of 100 runs were finished within 650 s. With this adjusted minsup, MSPS outperformed GSP (taking 2359 s) by a factor of about 3. This result demonstrates that our proposed formula for the adjusted minsup value is very reasonable.

As we can see in Table 4, when the user-specified minsup was lowered to 0.17% to mine the sample, the average performance was much worse than other adjusted minsup values. If it is lowered further, mining the sample itself becomes very hard and more overestimates will occur. When we increased the minsup to 0.5%, which was relatively too big, MSPS mined the sample and verified the sample result within just 11 s. Actually, it did not find long frequent sequences in the sample that could speed up the mining process. In such cases, MSPS works without the benefit of the supersequence frequency-based pruning. The performance gain comes mainly from the efficient counting of the candidates using the prefix tree structure and the customer sequence trimming. Compared to this case, the adjusted minsup of 0.20% reduced the total execution time by 34%, from 947 to 622 s.

Our additional tests showed that if the user-specified minsup is big, then it is better to use it without any change, or even a lowered one, for mining the sample. In Toivonen [18], how to avoid the misses in mining the sample by lowering the user-specified minsup is described. However, as mentioned before, without knowing whether a user-specified minsup is big or not for the database, lowering it to mine the sample could be very risky. In our research, we simulated the practical situations and adopted a safe approach of adjusting the minsup to a slightly bigger value by using the proposed formula to mine the sample. Our method is more useful when the user-specified minsup is small, and it is the case the mining process takes a lot of time.

7 Conclusions and future work

In this paper, we proposed a new algorithm, named MSPS, for mining maximal frequent sequences using sampling. MSPS combined the subsequence

infrequency-based pruning and the supersequence frequency-based pruning together to reduce the search space. In MSPS, a sampling technique is used to identify potential long frequent patterns early. When the user-specified minsup is small, we proposed how to adjust it to a little bigger value for mining the sample to avoid many overestimates. This method makes the sampling technique more efficient in practice for sequence mining. Both the supersequence frequency-based pruning and the customer sequence trimming used in MSPS improve the candidate counting process on the new prefix tree structure developed. Our extensive experiments proved that MSPS is a practical and efficient algorithm. Its excellent scalability makes it a very good candidate for mining customer market-basket databases which usually have tens of thousands of items and millions of customer sequences.

There are some research topics that need to be investigated further. (1) Theoretically analyzing the side-effect of sampling on the maximal sequential patterns. (2) Improving the sample quality: if a random sample does not represent the content of the original database well, the performance of MSPS is affected. We are considering the combination of sequence clustering and stratified sampling to improve the sample quality. (3) Integrating the proposed sampling technique with other sequence mining algorithms. It can improve their performance in certain application domains. (4) Investigating if there is a cost-effective way, other than sampling, that can detect long potential frequent sequences. If yes, combining it with the depth-first search could be an interesting topic.

Acknowledgements This research was supported in part by Ohio Board of Regents, NCR, and AFRL/Wright Brothers Institute (WBI).

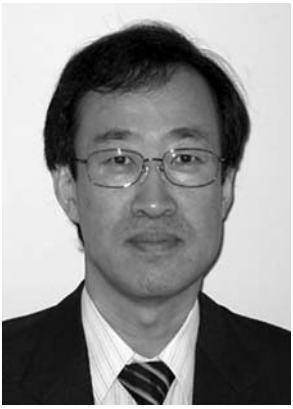
References

1. Agarwal RC, Aggarwal CC, Prasad VVV (2000) Depth first generation of long patterns. In: Proceedings of the 6th ACM SIGKDD international conference on knowledge discovery and data mining, pp 108–118
2. Agrawal R, Srikant R (1994) Fast algorithms for mining association rules. In: Proceedings of the 20th VLDB conference, pp 487–499
3. Agrawal R, Srikant R (1995) Mining sequential patterns. In: Proceedings of the international conference on data engineering, pp 3–14
4. Ayres J, Gehrke J, Yiu T, Flannick J (2002) Sequential pattern mining using a bitmap representation. In: Proceedings of the ACM SIGKDD international conference on knowledge discovery and data mining, pp 429–435
5. Bayardo RJ (1998) Efficient mining long patterns from databases. In: Proceedings of the ACM SIGMOD international conference on management of data, pp 85–93
6. Burdick D, Calimlim M, Gehrke J (2001) MAFIA: a maximal frequent itemset algorithm for transaction databases. In: Proceedings of the international conference on data engineering, pp 443–452
7. Chen B, Haas P, Scheuermann P (2002) A new two-phase sampling based algorithm for discovering association rules. In: Proceedings of the ACM SIGKDD international conference on knowledge discovery and data mining, pp 462–468
8. Chiu D, Wu Y, Chen ALP (2004) An efficient algorithm for mining frequent sequences by a new strategy without support counting. In: Proceedings of the international conference on data engineering, pp 375–386
9. Chung SM, Luo C (2004) Distributed mining of maximal frequent itemsets from databases on a cluster of workstations. In: Proceedings of the 4th IEEE/ACM international symposium on cluster computing and the grid—CCGrid 2004

10. Domingo C, Gavalda R, Watanabe O (1999) On-line sampling methods for discovering association rules. Tokyo Tech Rep. C-126. Department of Math and Computing Science, Tokyo Institute of Technology, Tokyo, Japan
11. Domingo C, Gavalda R, Watanabe O (2002) Adaptive sampling methods for scaling up knowledge discovery algorithms. Data Min Knowl Discov 6(2):131–152
12. Masseglia F, Cathala F, Poncelet P (1998) The PSP approach for mining sequential patterns. In: Proceedings of the European symposium on principle of data mining and knowledge discovery, pp 176–184
13. Mendenhall W, Sincich T (1995) Statistics for engineering and the sciences, 4th edn. Prentice-Hall, Englewood Cliffs, NJ
14. Park JS, Chen MS, Yu PS (1997) Using a hash-based method with transaction trimming for mining association rules. IEEE Trans Knowl Data Eng 9(5):813–825
15. Pei J, Han J, Mortazavi-Asl B, Pinto H, Chen Q, Dayal U, Hsu MC (2001) PrefixSpan: mining sequential patterns efficiently by prefix-projected pattern growth. In: Proceedings of the international conference on data engineering, pp 215–224
16. Shintani T, Kitsuregawa M (1998) Mining algorithms for sequential patterns in parallel: hash based approach. In: Proceedings of the Pacific-Asia conference on research and development in knowledge discovery and data mining, pp 283–294
17. Srikant R, Agrawal R (1996) Mining sequential patterns: generalizations and performance improvements. In: Proceedings of the 5th international conference on extending database technology, pp 3–17
18. Toivonen H (1996) Sampling large databases for association rules. In: Proceedings of the 22nd VLDB conference, pp 134–145
19. Wang J, Han J (2004) BIDE: efficient mining of frequent closed sequences. In: Proceedings of the international conference on data engineering, pp 79–90
20. Yan X, Han J, Afshar R (2002) CloSpan: mining closed sequential patterns in large datasets. In: Proceedings of the SIAM international conference on data mining, pp 166–177
21. Yang J, Wang W, Yu PS, Han J (2002) Mining long sequential patterns in a noisy environment. In: Proceedings of the ACM SIGMOD international conference on management of data, pp 406–417
22. Zaki MJ, Parthasarathy S, Li W, Ogihara M (1997) Evaluation of sampling for data mining of association rules. In: Proceedings of the 7th international workshop on research issues in data engineering, pp 42–50
23. Zaki MJ (2001) SPADE: an efficient algorithm for mining frequent sequences. Mach Learn 42(1):31–60

Author Biographies

Congnan Luo received the B.E. degree in Computer Science from Tsinghua University, Beijing, P.R. China, in 1997, the M.S. degree in Computer Science from the Institute of Software, Chinese Academy of Sciences, Beijing, P.R. China, in 2000, and the Ph.D. degree in Computer Science and Engineering from Wright State University, Dayton, OH, in 2006. Currently he is a technical staff at the Teradata division of NCR in San Diego, CA, and his research interests include data mining, machine learning, and databases.



Soon M. Chung received the B.S. degree in Electronic Engineering from Seoul National University, Korea, in 1979, the M.S. degree in Electrical Engineering from Korea Advanced Institute of Science and Technology, Korea, in 1981, and the Ph.D. degree in Computer Engineering from Syracuse University, Syracuse, New York, in 1990. He is currently a Professor in the Department of Computer Science and Engineering at Wright State University, Dayton, OH. His research interests include database, data mining, Grid computing, text mining, XML, and parallel and distributed processing.