A Scalable Algorithm for Mining Maximal Frequent Sequences Using Sampling

Congnan Luo and Soon M. Chung*
Dept. of Computer Science and Engineering
Wright State University
Dayton, Ohio 45435, USA

Abstract

In this paper, we propose an efficient scalable algorithm for mining Maximal Sequential Patterns using Sampling (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, 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 performance. This method makes sampling more efficient when the minimum support is small. A signature technique is utilized 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.

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 was introduced in [2], 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 [8] and PSP [6], 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 supersequence is frequent. Mining only the the set of maximal frequent sequences (MFS) is efficient because the search space can be reduced a lot by using the supersequence frequency based pruning. In interactive data mining, after mining the set of maximal frequent sequences quickly, we can selectively count the interesting patterns subsumed by this set by scanning the database just once.

For the association rule mining, many efficient algorithms were proposed to mine maximal frequent itemsets [5]. However, differences between the two kinds of mining make those algorithms very difficult or impossible to be applied for the maximal frequent sequence mining. A critical question for the maximal frequent sequence mining is how to look ahead for longer or maximal frequent sequences at a reasonable cost. If the look-ahead is not performed effectively, its cost can offset the gain from the supersequence frequency based pruning, like the cases of AprioriSome and DynamicSome algorithms [2].

In this research, we combined the Apriori candidate generation method [1, 2, 8] and the supersequence frequency pruning for mining maximal frequent sequences. This is achieved by using a sampling technique. The main search strategy of MSPS 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 3 items) generated from L_2, the set of frequent 2-sequences. The local maximal frequent sequences, that are found from the 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 supersequence frequency based pruning is applied at each pass.

The main contributions of this research are: 1) A new

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

Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004)
1082-3409/04 $20.00 © 2004 **IEEE**

MSPS algorithm and optimization methods were developed for mining maximal frequent sequences. MSPS outperforms GSP considerably, and it also shows better scalability than SPAM [3] and SPADE [11]. 2) In our research, how to apply the sampling technique for sequence mining was studied thoroughly. In association rule mining using sampling, lowering the user-specified minimum support (denoted by *minsup*) further was proposed for mining a sample to guarantee no misses (i.e., false negatives). For the case of sequence mining, where the search space is much larger, if the user-specified minsup is small, simply using this minsup or a lowered one often misleads the mining on a sample, such that the cost of mining the sample itself and verifying the sample results could be too high. For MSPS, we proposed a theoretic method of adjusting the small user-specified minsup to avoid this problem.

The rest of the paper is organized as follows. Section 2 introduces the basic concepts of sequence mining. Section 3 reviews some related works on sequence mining and sampling. Section 4 describes the MSPS algorithm. The experimental results and performance analyses are presented in Section 5, and Section 6 contains some conclusions.

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 $1 \leq m_1 < m_2 < \ldots < m_k \leq n$. A sequence s is an ordered list of itemsets denoted by $< s_1, s_2, \ldots, s_k >$, where each s_i, $1 \leq i \leq k$, is an *itemset*. A sequence $s_a =< a_1, a_2, \ldots, a_p >$ is contained in another sequence $s_b =< b_1, b_2, \ldots, b_q >$ if there exist integers $1 \leq j_1 < j_2 < \ldots < j_p \leq q$ 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 $< \{A\}, \{B, C\} >$ is a subsequence of a 5-sequence $< \{C\}, \{A, D\}, \{B, C\} >$. 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 which 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 $< T_1, T_2, \ldots, T_m >$, 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 this 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.

In [8], the above definition of sequence mining was generalized by incorporating time constraints, sliding time windows, and taxonomy. This generalization makes the sequence mining more complex. For example, a sequence $A - BC - D - GH$ is frequent does not necessarily mean that its subsequence $A - BC - GH$ is also frequent, because the subsequence may not satisfy the time constraints. In this research, we consider the nongeneralized sequential pattern discovery.

3 Related Work

Mining sequential patterns was introduced in [2] 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 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 [8] was proposed for generalized sequence mining. GSP requires multiple passes on the database. At pass k, the set of candidate k-sequences are counted against 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 [6]. For PSP, a prefix tree was developed as the internal data structure to organize and count candidates more efficiently.

SPADE [11] works on the databases with a vertical id-

list format, where a list of (customer-id, transaction-time) pairs are associated with each item, 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 [7] 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 pseudo-projection technique is developed to save the computation cost of projection and the memory space for projected databases. SPAM [3] 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.

These three algorithms (SPADE, PrefixSpan, and SPAM) 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 a vertical one requires extra disk space with roughly the same 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 one on the fly. This usually requires much time and memory for very large databases and results in performance degradation.

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, 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. SPAM 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.

In [10], sampling was evaluated as an efficient way to mine an approximate set of frequent itemsets. In [4], the FAST algorithm mines a large sample first to accurately estimate the support of 1-itemsets, and then progressively refine the initial sample to obtain a small final sample. FAST reports the set of frequent itemsets in the final sample as the results. Our research is more related to [9] because both try to speed up the mining of the exact final result using sampling. In [9], a lowered minsup is used to mine the sample, so that the probability a frequent itemset is missed from the sample result would be small. Then, one more database scan is needed to find the misses and the overestimates. In [9], the focus was mainly on how to make the sample result include all frequent itemsets without missing. On the other hand, our main goal is not the sample result itself, but obtaining some knowledge that can speed up the mining of exact final result. Thus, the misses are allowed to some extent. In practice, the cost for avoiding the misses can be a concern. Therefore, a critical question is how much sampling can speed up the mining of the exact result, and we explored this question from the point of balancing the cost and the gain of sampling.

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 technique is used 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 theoretic method of adjusting the small user-specified minsup for mining the sample database was 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. 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 sets of all local frequent k-sequences, candidate k-sequences and maximal frequent sequences in the sample, respectively.

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 two-dimensional array is used. The entry at position (i, j) in the upper-triangle of the array 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 for long global frequent sequences from 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.

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 $LongFS^{DB}$ don't 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.

4.2 Candidate Generation and Pruning

Both in 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:

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 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 in s_2. The added item starts a new itemset for s_1 if it was a separate itemset in s_2. Otherwise, it becomes part 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 deleted. 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 using a relational merge-join technique to generate candidates. But in this manner, 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 frequent swaps. Without subsequence infrequency based pruning, usually the performance of GSP degrades a lot. In MSPS, we adopted a new method to solve this problem. If some L_k^{db} in Phase 2 (L_k^{DB} in phase 3) requires too much memory, we give each local (global) frequent k-sequence an integer signature which is highly correlated to the content of the sequence. Following is a simple example of the signature, where t is the number of itemsets in the sequence; m_i is the number of items in ith itemset; I_{ij} is the jth item in the ith itemset; C_i, $1 \leq i \leq t$, is the weight imposed on the ith itemset; and C_0 is the weight imposed on the total number of itemsets.

$$(C_0 * t) + \sum_{i=1}^{t}(C_i * m_i * \sum_{j=1}^{m_i} I_{ij})$$

All the signatures are sorted and put into an array. Compared with the case of loading the whole L_k^{db} (L_k^{DB}) into memory, the signature array requires much less space. Thus, we can load working portions of L_k^{db} (L_k^{DB}) and all the signatures into memory at the same time. When a new candidate $(k+1)$-sequence is generated, the signatures of its k-subsequences are computed and searched in the signature array. If any one of them is not in the array, the candidate should be removed. Since all the signatures are in memory, subsequence infrequency pruning can still be applied. It is possible that two or more k-subsequences have the same signature. However, that probability is very low. Our experiments showed that signatures are much more efficient than

hashing. MSPS performs much better than GSP when the seed set of frequent sequences for generating the candidates cannot be loaded into memory totally at some passes. If the memory cannot hold all the candidates generated, they can be processed part by part.

4.3 Counting 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 [6].

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 Figure 1. Each 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 pathes represent the candidates $A-A$ and $A-E$, respectively. Nodes 6 and 7 are I-extension children, and their pathes 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 8 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 counting. Thus, the bit vector is set as (10111001), where 1 at the i-th 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 trim it to $s' = ACD - ADE - DH$ using the bit vector first. 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

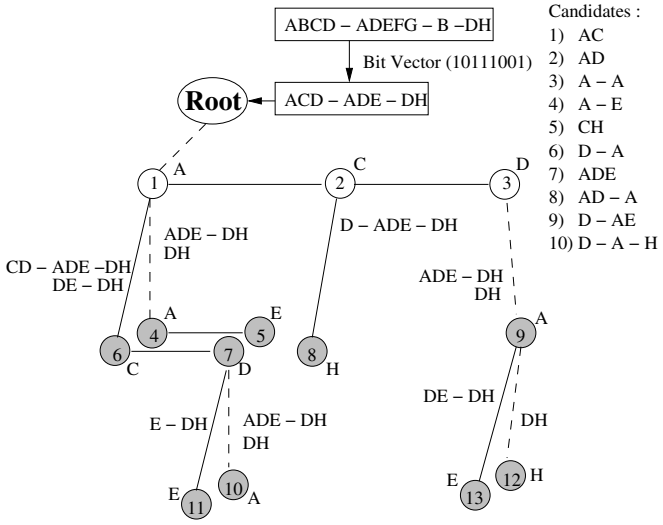


Figure 1. Prefix Tree of MSPS

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 it 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 always some constraints. Let's 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 should check the items in their first itemsets, CD and DE, against node 1's I-extension link. 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. During the candidate counting, we frequently need to locate the items in the customer sequences along these links.

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 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.

4.4 Analysis of the Sampling

Here, we discuss some important issues in sampling. 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*. On the other hand, if a pattern is infrequent in db but actually frequent in DB, it is a *miss*.

Both our research and [9] 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 [9] on how to reduce the probability of misses. To achieve that goal, two methods were suggested in [9]: 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 results, 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.

4.4.1 Sample Size

In [9, 10], the minimum sample size that guarantees a small chance of misses with certain confidence is given by the Chernoff boundary. Unfortunately, this theoretic guideline is not quite practical because it is too conservative. In MSPS, a large sample can improve the quality of sample results 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. On the other hand, 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., $minsup * |db|$ or $lowered\_minsup * |db|$) 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's why the sample should not be too large or too small.

In general, we do not know the 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.

4.4.2 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 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 bigger. For MSPS, it not only degrades the efficiency of mining the sample, but also causes a high cost to identify the overestimates.

In [9], 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 $lowered\_minsup * |db|$ or $minsup * |db|$ 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 2) and 3) cases 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. On the other hand, if it is increased too much, we may not find any long patterns from the sample.

Consider the original database DB and an arbitrary sequence X. If the support of X in DB is P_X, then the probability that a customer sequence randomly selected from DB contains X is also P_X. Let's consider a random sample S with m 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 S, has a binomial distribution of m trials with the probability of success P_X. In general, if m is greater than 30, T_X can be approximated by a normal distribution whose mean is $m * P_X$ and the standard deviation is $\sqrt{m * P_X * (1 - P_X)}$.

In MSPS, suppose that we draw a sample S with m customer sequences from DB, and then try to use the point estimator $P'_X = T_X/m$ to estimate the support of X in the population of DB. Then, P'_X is an unbiased estimator with mean $m * P_X/m = P_X$ and standard deviation $\sqrt{m * P_X * (1 - P_X)}/m = \sqrt{P_X * (1 - P_X)/m}$.

If we assume that the support of X in DB, P_X, is the user-specified minsup that we want to estimate, then P'_X, which is observed from a sample S, should be around P_X with a normal distribution as described above. If the adjusted minsup is denoted as P''_X, we can assume $P''_X > P_X$ because our goal is to find out how much we should increase the minsup for mining the sample. If we use P''_X to mine the sample, the probability that the sequence X can be found as a local frequent sequence in db is $1 - P_Z$, where $Z = (P''_X - P_X)/\sqrt{P_X * (1 - P_X)/m}$, and it is often called the z-score.

Let's consider the standard deviation of P'_X, $\sqrt{P_X * (1 - P_X)/m}$. The value of its part $P_X * (1 - P_X) = -(P_X - 1/2)^2 + 1/4$ is increasing in the P_X interval of $[0, 1/2]$. Since minsup is usually smaller than 50%, we can assume the value of $P_X * (1 - P_X)$ is increasing in the P_X interval of $[0, minsup]$; that means, the standard deviation of P'_X is increasing in this P_X interval. If the support of another sequence Y in DB is lower than the minsup P_X,

then both the mean and the standard deviation of observed P'_Y should be smaller than those of P'_X, respectively. Therefore, compared with P'_X, the distribution curve of P'_Y is shifted left and is shaper. Thus, if we set the adjusted minsup for mining the sample as P''_X, the probability that an infrequent sequence Y is identified as frequent in S should be lower than $1 - P_Z$. This guarantees the probability that any infrequent sequence is identified as frequent 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%. From $Z = (P''_X - P_X)/\sqrt{P_X * (1 - P_X)/m}$, we can drive $P''_X = P_X + Z * \sqrt{P_X(1 - P_X)/m}$, where $P_X = minsup$ and $m = |db|$.

This formula provides a theoretic guideline for adjusting the user-specified minsup to mine the sample. Even though this adjusted minsup value may not be the best one all the time, it worked well in most of our experiments.

5 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 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 [2]. The database generation parameters are described in Table 1. For all databases, $N_S = 5000$ and $N_I = 25,000$; and the names of these databases reflect other parameter values used to generate them.

5.1 Performance Comparison

We ran MSPS, GSP, SPADE and SPAM on three 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 memory, hence caused the termination.

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

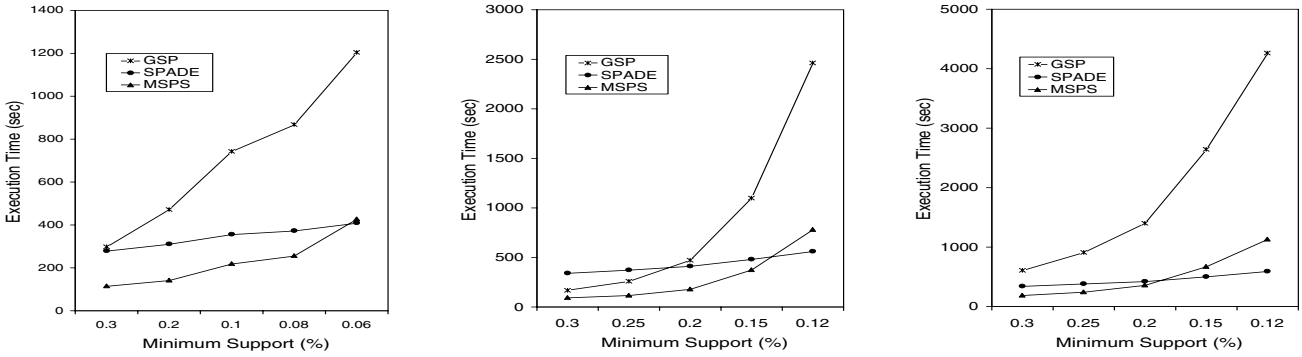
As discussed before, when the user-specified minsup is small, simply using it or a lowered one to mine the sample may cost too much due to so many overestimates. In practice, we may not know the data distribution characteristics of the database to be mined. Thus, we conservatively assumed that all user-specified 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 given before. The probability that an overestimate occurs is set to 10% at most, i.e., $Z = 1.28$.

The test results are shown in Figure 2. 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 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 the 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. 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.

5.2 Scalability Evaluation

Both SPADE and SPAM need to store a huge amount of intermediate data to save their computation cost. When



(a) D400K-C10-T5-S5-I1.25-N10K (b) D400K-C20-T2.5-S10-I2.5-N10K (c) D400K-C20-T2.5-S10-I1.25-N10K

Figure 2. Performance Comparison on Medium Size Databases

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} can not be totally loaded into memory by using the signatures as explained in Section 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 keep the user-specified minsup unchanged to mine it. 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 $D * C * N/8$ bytes. For the id-lists in SPADE, it is about $D * C * T * 4$ bytes. But we found these values are usually far less from their peak memory space requirement during the mining, because the amount of intermediate data in both algorithms is quite huge.

Compared with SPAM, SPADE divides search space into small pieces so that only the id-lists being processed need to be loaded into memory. Another advantage of SPADE is that the id-lists become 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.

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 400,000 to 2,000,000. 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 400,000 customers so that it is only about 100 Mbytes, which is one tenth of our main memory size. Figure 3 shows that the scalability of MSPS and GSP are quite linear. As the database size is increased, MSPS performs much better than the others.

When database is relatively small with only 400,000 customers, SPADE performed the 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 transformation 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.

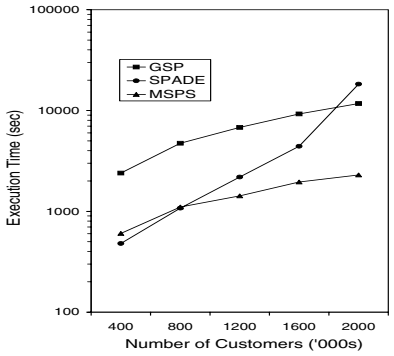


Figure 3. Scalability: Number of Customers (on C10-T5-S10-I2.5-N10K, minsup=0.18%)

Finally, we mined a large database D2000K-C10-T5-S10-I2.5-N10K, which takes about 500 Mbytes, for various minsups. This database is partitioned into 5 chunks for SPADE, and the results are shown in Figure 4.

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 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, others due to its simplicity and effective subsequence infrequency based pruning.

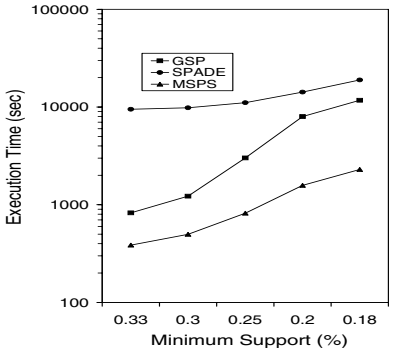


Figure 4. Performance on a Large Database D2000K-C10-T5-S10-I2.5-N10K

6 Conclusions

In this paper, we proposed a new algorithm 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.

References

[1] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," *Proc. of the 20th VLDB Conf.*, 1994, pp. 487–499.

[2] R. Agrawal and R. Srikant, "Mining Sequential Patterns," *Proc. of Int'l Conf. on Data Engineering*, 1995, pp. 3–14.

[3] J. Ayres, J. Gehrke, T. Yiu, and J. Flannick, "Sequential Pattern Mining Using a Bitmap Representation," *Proc. of ACM SIGKDD Conf. on Knowledge Discovery and Data Mining*, 2002, pp. 429–435.

[4] B. Chen, P. Haas, and P. Scheuermann, "A New Two-Phase Sampling Based Algorithm for Discovering Association Rules," *Proc. of ACM SIGKDD Conf. on Knowledge Discovery and Data Mining*, 2002, pp. 462–468.

[5] S. M. Chung and C. Luo, "Efficient Mining of Maximal Frequent Itemsets from Databases on a Cluster of Workstations," to appear in *IEEE Transactions on Parallel and Distributed Systems*.

[6] F. Masseglia, F. Cathala, and P. Poncelet, "The PSP Approach for Mining Sequential Patterns," *Proc. of European Symp. on Principle of Data Mining and Knowledge Discovery*, 1998, pp. 176–184.

[7] 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. of Int'l. Conf. on Data Engineering*, 2001, pp. 215–224.

[8] R. Srikant and R. Agrawal, "Mining Sequential Patterns: Generalizations and Performance Improvements," *Proc. of the 5th Int'l Conf. on Extending Database Technology*, 1996, pp. 3–17.

[9] H. Toivonen, "Sampling Large Databases for Association Rules," *Proc. of the 22nd VLDB Conf.*, 1996, pp. 134–145.

[10] M. J. Zaki, S. Parthasarathy, W. Li, and M. Ogihara, "Evaluation of Sampling for Data Mining of Association Rules," *Proc. of the 7th Int'l Workshop on Research Issues in Data Engineering*, 1997.

[11] M. J. Zaki, "SPADE: An Efficient Algorithm for Mining Frequent Sequences," *Machine Learning*, 42(1), 2001, pp. 31–60.