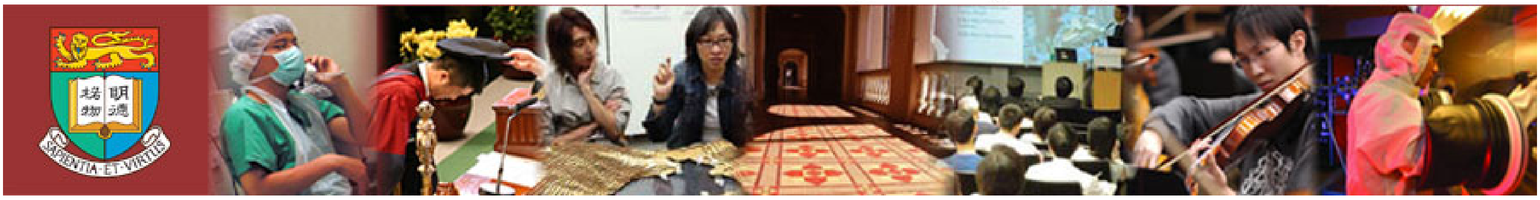

A Comparison Study on Algorithms for Incremental Update of Frequent Sequences *

Minghua Zhang   Ben Kao   Chi-Lap Yip

Department of Computer Science and Information Systems,
The University of Hong Kong, Hong Kong.
{mhzhang, kao, clyip}@csis.hku.hk

Abstract

The problem of mining frequent sequences is to extract frequently occurring subsequences in a sequence database. Algorithms on this mining problem include GSP, MFS, and SPADE. The problem of incremental update of frequent sequences is to keep track of the set of frequent sequences as the underlying database changes. Previous studies have extended the traditional algorithms to efficiently solve the update problem. These incremental algorithms include ISM, GSP+ and MFS+. Each incremental algorithm has its own characteristics and they have been studied and evaluated separately under different scenarios. This paper presents a comprehensive study on the relative performance of the incremental algorithms as well as their non-incremental counterparts. Our goal is to provide guidelines on the choice of an algorithm for solving the incremental update problem given the various characteristics of a sequence database.

keyword: sequence, incremental update, data mining

1. Introduction

One of the many data mining problems is mining frequent sequences from transactional databases. The goal is to discover frequent sequences of events. The problem was first introduced by Agrawal and Srikant [1]. In their model, a database is a collection of transactions. Each transaction is a set of items (or an itemset) and is associated with a customer ID and a time ID. If one groups the transactions by their customer IDs, and then sorts the transactions of each group by their time IDs in increasing value, the database is transformed into a number of customer sequences. Each customer sequence shows the order of transactions a customer has conducted. Roughly speaking, the problem of mining frequent sequences is to discover "subsequences" (of itemsets) that occur frequently enough among all the customer sequences.

Sequence mining finds its application in many different areas. A few efficient algorithms for mining frequent sequences have been proposed, notably, AprioriAll [1], GSP [6], SPADE [7], MFS [9] and PrefixSpan [4].

We note that in a typical data mining process, data is rarely fully collected in one attempt. In many cases, data collection is carried out in phases. Consequently, the content of the underlying database changes over time. To keep track of the frequent sequences, sequence mining algorithms have to be executed whenever the underlying database changes. We refer to this problem the incremental update of frequent sequences.

A simple approach to the update problem is to mine the new database from scratch, using a sequence mining algorithm mentioned previously. This approach, however, fails to take advantage of the valuable information obtained from the mining results of a previous exercise. To utilize the previous mining results to efficiently mine the updated database, several incremental versions of the basic sequence mining algorithms have been proposed, including GSP+ [8], MFS+ [8] and ISM[3]. The above three incremental algorithms are based on GSP, MFS and SPADE, respectively.

With the different algorithms for (incremental) sequence mining available, an interesting question is which one to choose given a particular application with a specific dataset characteristic and a particular computing environment. We note that the incremental algorithms we just mentioned, namely, GSP+, MFS+, and ISM, have different requirements and assumptions. They were also studied and evaluated separately under different scenarios.

Our goal in this paper is to conduct a comprehensive study on the relative performance of the algorithms. We evaluate the three incremental algorithms (GSP+, MFS+, and ISM) as well as their non-incremental counterparts

*This research was supported by Hong Kong Research Grants Council under grant number HKU 7040/02E.

0-7695-1754-4/02 $17.00 © 2002 IEEE

(GSP, MFS, and SPADE). This comparison study allows us to:

- identify the various factors (such as database update model, data characteristics, and memory availability) that affect the algorithms' performance;
- understand when and how an incremental algorithm outperforms its non-increment version; and
- provide guidelines on how to choose the most efficient algorithm.

The rest of this paper is organized as follows. In Section 2 we give a formal definition of the problem. We also states the two models of database update, namely, the sequence model and the transaction model. In Section 3 we briefly describe the six algorithms: GSP, MFS, SPADE, GSP+, MFS+, and ISM. Section 4 presents an extensive performance study. Finally, we conclude the paper in Section 5.

2. Problem definition and model

In this section, we give a formal definition of the problem of mining frequent sequences. Also, we describe two update models of the incremental update problem.

2.1. Mining frequent sequences

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of literals called items. An itemset X of the universe I is a set of items. A sequence $s = \langle t_1, t_2, \ldots, t_n \rangle$ is an ordered set of transactions, where each transaction t_i $(i = 1, 2, \ldots, n)$ is an itemset.

The length of a sequence s is defined as the number of items contained in s. If an item occurs several times in different itemsets of a sequence, the item is counted for each occurrence. We use $|s|$ to represent the length of s.

Given two sequences $s_1 = \langle a_1, a_2, \ldots, a_n \rangle$ and $s_2 = \langle b_1, b_2, \ldots, b_l \rangle$, we say s_1 contains s_2 (or equivalently s_2 is a subsequence of s_1) if there exist integers $j_1, j_2, \ldots, j_l$, such that $1 \leq j_1 < j_2 < \ldots < j_l \leq n$ and $b_1 \subseteq a_{j_1}, b_2 \subseteq a_{j_2}, \ldots, b_l \subseteq a_{j_l}$. We represent this relationship by $s_2 \sqsubseteq s_1$.

In a sequence set V, a sequence $s \in V$ is *maximal* if s is not a subsequence of any other sequence in V.

Given a database D of sequences, the *support count* of a sequence s, denoted by δ_D^s, is defined as the number of sequences in D that contain s. The *fraction* of sequences in D that contain s is called the *support* of s. If we use the symbol $|D|$ to denote the number of sequences in D (or the size of D), then support of $s = \delta_D^s / |D|$.

If the support of s is not less than a user specified support threshold, ρ_s, s is a frequent sequence. The problem of mining frequent sequences is to find all *maximal* frequent sequences in a database D.

2.2. Update model

A sequence database can be updated in two ways, depending on whether a sequence or a transaction is the unit of update. In this subsection, we describe the two update models.

Sequence Model. In the sequence model, the database is updated by adding and/or removing whole sequences.

Formally, in the sequence model, we assume that a previous mining exercise has been executed on a sequence database D to obtain the support counts of its frequent sequences. The database D is then updated by deleting a set of sequences Δ^- followed by inserting a set of sequences Δ^+. Let us denote the updated database by D'. Note that $D' = (D - \Delta^-) \cup \Delta^+$. We denote the set of unchanged sequences by $D^- = D - \Delta^- = D' - \Delta^+$.

For the incremental update problem, the objective is to find all maximal frequent sequences in the database D' given Δ^-, D^-, Δ^+, and the mining result of D.

We note that in [8], algorithms GSP+ and MFS+ are evaluated under the sequence model.

Transaction Model. In the transaction model of the update problem, individual sequences in the database can be updated by appending new transactions.

Given a sequence database D and its mining result, and a set of transactions ΔT, the incremental update problem under the transaction model is to determine the set of maximal frequent sequences in D with the transactions in ΔT appended to the sequences in D. In [3], algorithm ISM is evaluated under the transaction model.

Before we end this section, we remark that technically the two update models can model each other. For example, if transactions are appended to a sequence s to form a new sequence s', we can consider this update as a removal of s from the database followed by an insertion of s'. Also, if a sequence s is inserted into a database, we can consider this update as appending all the transactions in s to an initially empty sequence.[1] In fact, the studies on the three incremental algorithms claim that the algorithms are applicable under both update models by the above mentioned mapping. One of the goals of this paper is to determine the effectiveness of these algorithms under the different update models.

3. Algorithms

In this section, we review three sequence mining algorithms, namely, GSP, MFS, SPADE, and their incremental versions, namely, GSP+, MFS+, and ISM.

[1] The transaction model cannot model sequence deletion unless we consider transaction removal as well. We do not consider this modification in this paper.

3.1. GSP

Algorithm GSP was proposed by Srikant and Agrawal [6]. Similar to the structure of the Apriori algorithm [5] for mining association rules, GSP starts by finding all frequent length-1 sequences from the database. A set of candidate length-2 sequences is then generated. The support counts of the candidate sequences are then counted by scanning the database once. Those frequent length-2 sequences are then used to generate candidate sequences of length 3, and so on. In general, GSP uses a function GGen to generate candidate sequences of length $k+1$ given the set of all frequent length-k sequences. The algorithm terminates when no more frequent sequences are discovered during a database scan. For the details of the candidate generation function GGen, please refer to [6].

GSP is an efficient algorithm. However, the number of database scans it requires is determined by the length of the longest frequent sequences. Consequently, if there are very long frequent sequences and if the database is huge, the I/O cost of GSP could be substantial.

3.2. MFS

To improve the I/O efficiency of GSP, the algorithm MFS was proposed [9]. Similar to GSP, MFS is an iterative algorithm. MFS first requires an initial estimate, S_{est}, of the set of frequent sequences of the database be available. The set S_{est} can be obtained by mining a small sample of the database, and using the frequent sequences of the sample as S_{est}. For the incremental update problem, MFS can use the frequent sequences in the old database as S_{est}.

In the first iteration of MFS, the database is scanned to obtain the support counts of all length-1 sequences as well as those of the sequences in the estimated set, S_{est}. Sequences that are found frequent are collected into a set *MFSS*. Essentially, *MFSS* captures the set of frequent sequences that MFS has known so far. Typically, the set *MFSS* contains frequent sequences of various lengths. MFS then applies a candidate generation function MGen on *MFSS* to obtain a set of candidate sequences. The database is then scanned to determine which candidate sequences are frequent. Those that are frequent are added to the set *MFSS*. MFS then again applies the generation function MGen on the *refined MFSS* to obtain a new set of candidate sequences, whose supports are then counted by scanning the database, and so on. MFS executes this candidate-generation-verification-refinement iteration until the set *MFSS* cannot be refined further.

The heart of MFS is the candidate generation function MGen. MGen can be considered as a generalization of GGen (used in GSP) in that MGen takes as input a set of frequent sequences of various lengths and generates a set of candidate sequences of various lengths. For the details of the MGen function, please refer to [9].

It shows that if the set S_{est} is a reasonably good estimate of the true set of frequent sequences, then MFS will generate long candidate sequences early (compared with GSP). As a result, in many cases, MFS requires fewer database scans and less data processing than GSP does. This reduces both CPU and I/O costs.

3.3. GSP+ and MFS+

Based on GSP and MFS, two incremental algorithms GSP+ and MFS+ were proposed in [8]. In the study, the sequence model of database update is assumed.

The structures of GSP+ and MFS+ follow those of GSP and MFS. The major difference is that during each iteration, after a set of candidate sequences is generated, the incremental algorithms first deduce whether a candidate sequence s can be frequent by considering the mining result of the old database, and in some cases, s's support count w.r.t. Δ^+ and/or Δ^- (the updated portion of the database). If the candidate sequence s cannot be frequent, it is pruned from the candidate set.

Two lemmas are proposed in [8] to help GSP+ and MFS+ make the pruning decision. In the lemmas, the symbol b_X^s refers to an upper bound of the support count of a sequence s in a dataset X, and is calculated by $b_X^s = \min_{s'} \delta_X^{s'}$, where $(s' \sqsubseteq s) \wedge (|s'| = |s| - 1)$.

Lemma 1 *If a sequence s is frequent in D', then $\delta_D^s + b_{\Delta^+}^s \geq \delta_D^s + b_{\Delta^+}^s - \delta_{\Delta^-}^s \geq |D'| \times \rho_s$.*

Lemma 2 *If a sequence s is frequent in D' but not in D, then $b_{\Delta^+}^s \geq b_{\Delta^+}^s - \delta_{\Delta^-}^s \geq \delta_{\Delta^+}^s - \delta_{\Delta^-}^s > (|\Delta^+| - |\Delta^-|) \times \rho_s$.*

Lemma 1 applies to a candidate sequence s that is frequent w.r.t. the old database D, and Lemma 2 applies to candidates that are infrequent in D. For a candidate sequence s that cannot be pruned, its support count w.r.t. the new database is then calculated. For further details, please refer to [8].

GSP+ and MFS+ gain efficiency by avoiding processing D^-(the unchanged part of the database). If the database does not change greatly across an update, then D^- is relatively large compared with Δ^+ and Δ^-. The performance gain would then be substantial.

3.4. SPADE

The algorithms we have reviewed so far, namely, GSP, MFS, GSP+ and MFS+ assume a *horizontal database representation*. In this representation, each row in the database table represents a transaction. Each transaction is associated with a customer ID, a transaction timestamp, and an

Table 1. Horizontal database

Customer ID	Transaction timestamp	Itemset
1	110	A
1	120	B C
2	210	A
2	220	C D

Table 2. Vertical database

Item	Customer ID	Transaction timestamp
A	1	110
	2	210
B	1	120
C	1	120
	2	220
D	2	220

itemset. Table 1 shows an example of a database in the horizontal representation.

In [7], it is observed that a *vertical* representation of the database may be better suited for sequence mining. In the vertical representation, every item in the database is associated with an id-list. For an item a, its id-list is a list of (customer ID, transaction timestamp) pairs. Each such pair identifies a unique transaction that contains a. A vertical database is composed of the id-lists of all items. Table 2 shows the vertical representation of the database shown in Table 1.

In [7], the algorithm SPADE is proposed that uses a vertical database to mine frequent sequences. To understand SPADE, let us first define two terms: *generating subsequences* and *sequence id-list.*

Generating subsequences. For a sequence s such that $|s| \geq 2$, the two generating subsequences of s are obtained by removing the first or the second item of s.

Sequence id-list. Similar to the id-list of an item, we can also associate an id-list with a sequence. The id-list of a sequence s is a list of (Customer ID, transaction timestamp) pairs. If the pair (C, t) is in the id-list of a sequence s, then s is contained in the sequence of Customer C, and that the first item of s occurs in the transaction of Customer C at timestamp t. Table 3 shows the id-list of $\langle\{A\},\{C\}\rangle$.

We note that if id-lists are available, counting the supports of sequences is trivial. In particular, the support count of a length-1 sequence can be obtained by inspecting the vertical database. In general, the support count of a sequence s is given by the number of distinct customer id's in s's id-list. The problem of support counting is thus reduced to the problem of sequence id-list computation.

With the vertical database, only the id-lists of length-1 sequences can be readily obtained. The id-lists of longer sequences have to be computed. It is shown in [7] that the id-list of a sequence s can be computed easily by *intersecting* the id-lists of the two generating subsequences of s.

Table 3. ID-list of $\langle\{A\},\{C\}\rangle$

Customer ID	Transaction timestamp
1	110
2	210

Table 4. Horizontal database generated for computing L_2

Customer ID	(item, transaction timestamp) pairs
1	(A 110) (C 120)
2	(A 210) (C 220)

Here, we summarize the key steps of SPADE.

1. Find frequent length-1 sequences. This is done by scanning the id-lists of the items from the vertical database.

2. Find frequent length-2 sequences. Suppose there are M frequent items, then the number of candidate frequent length-2 sequences is $O(M^2)$. If the support counts of these length-2 sequences are obtained by first computing their id-lists using the intersection procedure, we have to access id-lists from the vertical database $O(M^2)$ times.[2] This could be very expensive.

Instead, SPADE solves the problem by building a horizontal database on the fly that involves only frequent items. In the horizontal database, every customer is associated with a list of (item, transaction timestamp) pairs. For each frequent item found in Step 1, SPADE reads its id-list from disk and the horizontal database is updated accordingly. For example, if the frequent items of our example database (Table 2) are A, C, then the constructed horizontal database is shown in Table 4. After obtaining the horizontal database, the supports of all candidate length-2 sequences are computed from it.

We remark that maintaining the horizontal database might require a lot of memory. This is especially true if the number of frequent items and the vertical database are large.

3. Find long frequent sequences. In step 3, SPADE generates the id-lists of long candidate sequences (those of length $\geq$ 3) by the intersection procedure. SPADE carefully controls the order at which candidate sequences (and their id-lists) are generated to keep the memory requirement at a minimum. For details, readers are referred to [7].

3.5. ISM

ISM is an incremental update algorithm based on SPADE. With ISM, the transaction model of database up-

[2] This is because computing the id-list of a length-2 sequence requires accessing the 2 id-lists of the 2 items involved.

date is assumed, although it also handles sequence insertion.

Similar to SPADE, ISM requires the availability of the vertical database. Besides that, it needs a lattice structure called *increment sequence lattice*, or ISL w.r.t. the old database D. A node in ISL represents either a frequent sequence, or a sequence in the negative border (In ISM, a sequence is called in the negative border if it is infrequent, and either its length is 1 or both of its two generating subsequences are frequent). The node also contains the support count of the sequence w.r.t. D. Edges in the ISL connect a sequence with its generating subsequences. ISM assumes that the ISL of the old database is available before the incremental update.

There are three key steps of ISM.

In the first step, ISM checks whether there are new sequences added to the old database D in the update. If there are, ISM computes the new support count threshold and adjusts ISL accordingly. In the adjustment, frequent sequences may remain frequent, be moved to the negative border, or be deleted from ISL. Also, sequences in the negative border may stay in the negative border or be removed.

In the second step, ISM updates support counts of the sequences in ISL. And the third step of ISM is to capture sequences that were not originally in ISL. Similar to the first step, both the second step and the third step need to process ISL. For further details, please refer to [3].

4. Experiment results and analysis

We performed a number of experiments comparing the performance of the three incremental algorithms GSP+, MFS+, ISM and their non-incremental counterparts GSP, MFS and SPADE. The non-incremental algorithms were executed directly on the updated database. For MFS, GSP+, MFS+, and ISM, we assume that the set of frequent sequences w.r.t the old database and their support counts are available. For MFS and MFS+, this set of frequent sequences is used as the estimated set S_{est} (see Section 3). Furthermore, for SPADE and ISM, the database is stored in the vertical representation. Also, for ISM, we assume that ISL w.r.t. the old database is available. The experiments were done under the two database update models. In this section we present some representative results.

The experiments were performed on synthetic databases generated by the sequence generator of the IBM Quest data mining project [2]. The generator takes a number of parameters as input. In our experiment, we let $N_s = 5,000$, $N_i = 25,000$, and use $C10T2.5S4I1.25$ settings. For the details of the parameters, please refer to [1].

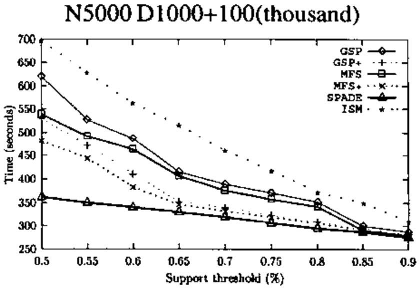

Figure 1. Execution time vs. support threshold (sequence model)

4.1. Sequence model

In the first set of experiments, the database is updated under the sequence model. We first generate a database D of 1,000,000 sequences. After that, another 100,000 sequences are generated and are inserted into the database to form a new database D'. In this experiment, the number of items, N, is set to 5,000. We execute SPADE on D to obtain the necessary information for the incremental algorithms. The six algorithms GSP, MFS, SPADE, GSP+, MFS+ and ISM are then executed to mine D'. The experiments were performed on a 700MHz PIII Xeon machine with 4GB of main memory running Solaris 8. The execution times of the six algorithms under different support thresholds ($0.5\% \leq \rho_s \leq 0.9\%$) are shown in Figure 1.

From the figure, we see that as the support threshold increases, the running times of all six algorithms decrease. This is because a larger ρ_s means fewer and shorter frequent sequences. Therefore, fewer iterations and less support countings are needed. Also, the performance difference among the algorithms is more substantial when the support threshold is small.

We observe that the two pruning algorithms, GSP+ and MFS+, perform much better than their non-incremental counterparts, GSP and MFS. As we have discussed, the savings mostly come from pruning candidate sequences. With fewer candidate sequences to consider, less amount of subsequence testing and support counting is done, which leads to performance gains. In general, MFS+ has a slight edge over GSP+ (and so does MFS over GSP). Recall that MFS+ (and also MFS) uses the set of frequent sequences w.r.t. the old database as an initial estimate (S_{est}). MFS+ is able to generate and count long sequences early, potentially reducing the I/O cost and the processing time.

Another interesting observation we can make from Figure 1 is that the incremental algorithm ISM performs worse than its non-incremental version, SPADE. This shows that ISM may not be a good choice under the sequence model of database update.

Recall that there are three key steps of ISM, all have to do with maintaining the increment sequence lattice (ISL). Moreover, before ISM terminates, it has to output ISL for the next incremental update.

We note that under the sequence model of database update, ISM needs to work harder in maintaining ISL compared with the case under the transaction model. First, under the transaction model of update, a sequence that is frequent w.r.t the old database must also be frequent w.r.t. the updated one; while it is not true under the sequence model. Also, the first step of ISM can be omitted under the transaction model, while this step of ISL adjustment is needed under the sequence model, since by inserting new sequences, the support count requirement is changed. So the changes made to ISL could thus be more drastic under the sequence model than it is under the transaction model. This explains why in our experiment, ISM performs worse than the other algorithms that do not handle ISL.

Finally, we see that SPADE is the most efficient algorithm. This shows that the vertical database representation allows very efficient support counting using the idea of idlists. A potential disadvantage of SPADE is that it requires much memory in the construction of a horizontal database (see Step 2 of the description of SPADE, Section 3.4). Since the machine on which we ran the experiment has 4GB of memory, the large memory requirement of SPADE is not a factor. We will study the impact of memory availability on SPADE later in this section.

The above discussion suggests that the performance of ISM is affected greatly by the size of ISL, which is in turn, dependent on a number of factors. One of these factors is the support threshold ρ_s. A larger ρ_s gives fewer frequent sequences, and a smaller negative border. Hence, ISL is smaller.

Another factor is the number of items, N, in the database. A large value of N is both a blessing and a curse. First, note that all length-1 sequences are in ISL Since each item derives one length-1 sequence, a large N gives a very fat (and large) ISL. On the other hand, if there are many items, transactions will have more variety. Given the same database parameters, there will be fewer frequent sequences. This factor makes ISL smaller. Figure 2 shows the performance of the algorithms under different values of N. Since GSP and MFS are outperformed by their incremental versions (i.e., GSP+ and MFS+), their curves are omitted to make the graph more readable. In this experiment, the support threshold is set to 0.7%. The number of items N is varied from 1,000 to 10,000.

From Figure 2, we see that, in general, when the number of items increases, the execution times of GSP+, MFS+ and SPADE decrease. This is because using the same values for the other parameters of the database generator, a larger number of items means each item has a smaller probability of appearing in a transaction. This leads to smaller support counts and thus fewer frequent sequences to discover. As a result, the three algorithms take less time to complete. For ISM, we see the opposing effects we mentioned earlier. When N is small, increasing N causes a dramatic increase in ISL's size, which outweights the effect of a reduction in the number of frequent sequences. The result is an increase in ISM's execution time. When N is large (say, $> 4,000$), the weightings of the two factors shift. This results in a decrease in execution time.

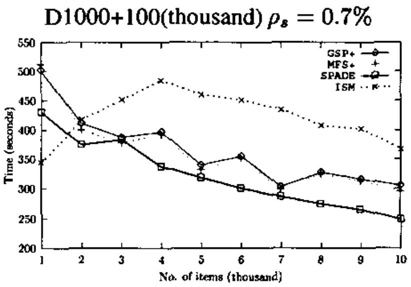

Figure 2. Execution time vs. N (sequence model)

From the figure, we notice that even under the sequence model, ISM can be the best algorithm. This happens when the number of items is very small. In this case, ISM maintains a small ISL, leading to a very efficient algorithm.

From Figures 1 and 2, we see that with a memory-abundant system (4GB in our experiment), SPADE is a very efficient algorithm. To study the effect of memory availability on the algorithms, we re-ran the experiment on an 866MHz PC with 512MB memory running Solaris 8. In the experiment, ρ_s is set to 0.7% and N is set to 5,000. We vary the size of D from 200,000 sequences to 2,000,000 sequences. For each case, the updated database D' is 10% larger than D. Figure 3 shows the result.

Figure 3 shows that while GSP+ and MFS+ scale linearly with the database size, SPADE and ISM perform poorly when the database is relatively large. The reason for the performance degradation is that SPADE requires a lot of memory to transform the vertical database to the horizontal one in order to find frequent length-2 sequences. When the database size is large compared with the amount of physical memory available, expensive memory paging occurs. Since ISM is based on SPADE, it suffers a similar performance degradation.

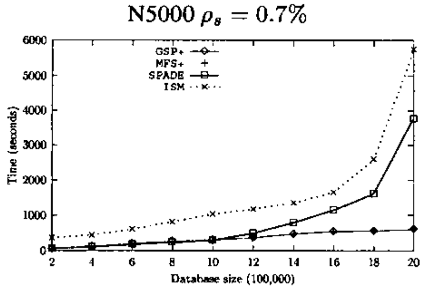

Figure 3. Effect of database size (sequence model)

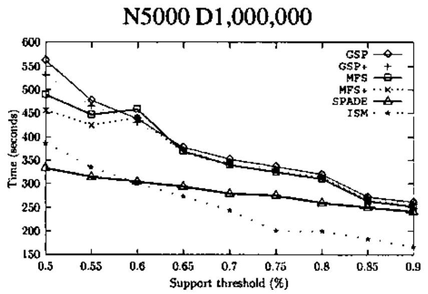

Figure 4. Execution time vs. support threshold (transaction model)

4.2. Transaction model

In the second set of experiments, we study the algorithms' performance under the transaction model of database update. In the experiments, we first generate a database of 1,000,000 sequences. We regard this database as the *updated* one, D'. We then randomly select 1% of the sequences from the database and delete the last two transactions from the selected sequences. We regard the resulting database as the old database, D. Hence, the update is equivalent to adding two transactions to 1% of the sequences in D. For GSP+ and MFS+, this update is modeled by sequence deletion followed by sequence insertion. We note that, in this case, $|\Delta^-| = |\Delta^+|$.

After the data generation, we run SPADE on the old database to obtain the necessary information for incremental algorithms. Figure 4 shows the performance of the six algorithms executed on a 700MHz Xeon machine with 4GB memory. In this experiment, the number of items, N, is set to 5,000, and the support threshold, ρ_s, is varied from 0.5% to 0.9%.

Similar to the sequence model, from Figure 4, we see that as ρ_s increases, in general, the execution times of the algorithms decrease. Again, this is because a larger ρ_s means fewer sequences to discover.

Unlike the sequence model case, under the transaction model, ISM performs better than the other algorithms (unless ρ_s is very small). This is because, under the transaction model of database update, there is much less change to ISL. For example, the first step of ISM is not needed. Moreover, ISM outperforms the other algorithms by a large margin when ρ_s is large. This is because ISL is small under a large ρ_s, hence its maintenance cost is small. On the other hand, when ρ_s is small, the set of frequent sequences as well as the negative border are large. In this case, ISM is not as efficient as SPADE, since it has to maintain a fairly large ISL.

From Figure 4, we also observe that under the transaction model, the two pruning algorithms (GSP+ and MFS+) are not very effective. They achieve very little performance gain over their non-incremental versions. Recall that the main idea of the pruning algorithms is to deduce which candidate sequences cannot be frequent without resorting to support counting. To make that deduction, the pruning algorithms consider two cases:

Case 1: a candidate sequence s is frequent w.r.t D. Under the transaction model, we note that a sequence that is frequent w.r.t D must also be frequent w.r.t D'. Hence, no sequences in this case can be pruned.

Case 2: a candidate sequence s is not frequent w.r.t. D. In this case, we check if the inequality $\delta^s_{\Delta^+} - \delta^s_{\Delta^-} > (|\Delta^+| - |\Delta^-|) \times \rho_s$ is true (see Lemma 2, Section 3.3). If not, s can be pruned. However, under the transaction model, $|\Delta^+| = |\Delta^-|$. Hence the right hand side of the inequality is always 0. The inequality is false only if $\delta^s_{\Delta^+}$ exactly equals $\delta^s_{\Delta^-}$, which is unlikely. Therefore, very few candidate sequences can be pruned.

With ineffective pruning, not much advantage is obtained from the pruning algorithms.

Finally, we remark that, with plenty of memory (4GB), SPADE performs consistently well over the range of ρ_s.

In another experiment, we study how the extent of database update affects the algorithms' performance. Again, we generate a database D' of 1,000,000 sequences. We then randomly select x% ($1 \leq x \leq 10$) of the sequences in D' from each of which the last two transactions are removed. The resulting database is used as D. Figure 5 shows the experiment result. For readability, we omit the curves for GSP and MFS, since their performance is very similar to that of GSP+ and MFS+.

From the figure, we see that GSP+ and MFS+ are relatively unaffected by the percentage change. The curve for SPADE stays flat since it is applied directly on the the updated database, and in the experiment D' stays the same. For ISM, its execution time increases linearly with the percentage change. This is because more update made to the database leads to more changes to ISL. ISM, therefore, has

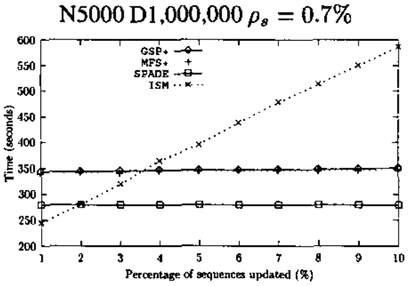

Figure 5. Execution time vs. percentage of sequences being updated (transaction model)

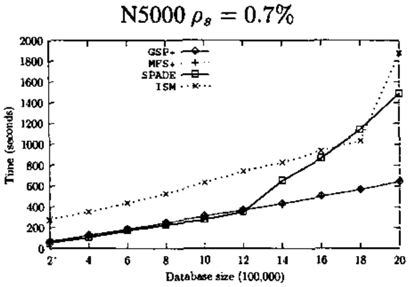

Figure 6. Execution time vs. database size (transaction model)

to spend more effort in updating the lattice. We see that for small database update (say, 1%), ISM gives the best performance. On the other hand, if the database is changed substantially, SPADE is the best choice.

Our last experiment studies the performance of the algorithms under the transaction model when memory is limited. We performed the experiment on an 866MHz PC with 512MB of memory. We varied the size of D' from 200,000 sequences to 2,000,000 sequences. In each run, 5% of the sequences in D' were selected to have their last two transactions removed to form D. Figure 6 shows the result. Again, we omit the curves of GSP and MFS for readability.

From the figure, we see that when the database is large, SPADE and ISM perform poorly. This is again because of their relatively large memory requirements.

5. Conclusions

In this paper we studied the problem of incremental update of frequent sequences. We compared the performance of three incremental algorithms, namely, GSP+, MFS+, ISM, and their non-incremental counterparts GSP, MFS and SPADE. We studied two database update models, namely, the sequence model and the transaction model. We discussed the various characteristics of the algorithms and showed their performance under various situations. Based on the experiment results, we derive the following guidelines on choosing the most efficient algorithm:

Under the sequence model of database update

- If the amount of main memory is relatively large compared with the database size and the number of items is small, ISM is the most efficient.
- If the amount of main memory is relatively large compared with the database size and the number of items is large, SPADE is the best choice.
- If memory is limited, GSP+ or MFS+ should be considered.

Under the transaction model of database update

- If memory is abundant and only a small portion of the database is updated, ISM is the best choice.
- If memory is abundant and a significant portion of the database is changed, SPADE is the most efficient.
- If the database is large compared with the amount of memory available, pick anyone of GSP, MFS, GSP+, or MFS+.

References

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. of the 11th Int'l Conference on Data Engineering*, Taipei, Taiwan, March 1995.

[2] http://www.almaden.ibm.com/cs/quest/.

[3] S. Parthasarathy, M. J. Zaki, M. Ogihara, and S. Dwarkadas. Incremental and interactive sequence mining. In *Proceedings of the 1999 ACM 8th International Conference on Information and Knowledge Management (CIKM'99)*, Kansas City, MO USA, November 1999.

[4] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Prefixspan: Mining sequential patterns by prefix-projected growth. In *Proc. 17th IEEE International Conference on Data Engineering (ICDE)*, Heidelberg, Germany, April 2001.

[5] T. I. R. Agrawal and A. Swami. Mining association rules between sets of items in large databases. In *Proc. ACM SIGMOD International Conference on Management of Data*, page 207, Washington, D.C., May 1993.

[6] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. of the 5th Conference on Extending Database Technology (EDBT)*, Avignion, France, March 1996.

[7] M. J. Zaki. Efficient enumeration of frequent sequences. In *Proceedings of the 1998 ACM 7th International Conference on Information and Knowledge Management(CIKM'98)*, Washington, United States, November 1998.

[8] M. Zhang, B. Kao, D. Cheung, and C.-L. Yip. Efficient algorithms for incremental update of frequent sequences. In *Proc. of the sixth Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, Taiwan, May 2002.

[9] M. Zhang, B. Kao, C. Yip, and D. Cheung. A GSP-based efficient algorithm for mining frequent sequences. In *Proc. of IC-AI'2001*, Las Vegas, Nevada, USA, June 2001.