SEQUEST: mining frequent subsequences using DMA-Strips

H. Tan[1], T. S. Dillon[1], F. Hadzic[1] & E. Chang[2]
[1]*Faculty of IT, University of Technology Sydney, Sydney, Australia*
[2]*School of IT, Curtin University, Perth, Australia*

Abstract

Sequential patterns exist in data such as DNA string databases, occurrences of recurrent illness, etc. In this study, we present an algorithm, SEQUEST, to mine frequent subsequences from sequential patterns. The challenges of mining a very large database of sequences is computationally expensive and require large memory space. SEQUEST uses a Direct Memory Access Strips (DMA-Strips) structure to efficiently generate candidate subsequences. DMA-Strips structure provides direct access to each item to be manipulated and thus is optimized for speed and space performance. In addition, the proposed technique uses a hybrid principle of frequency counting by the vertical join approach and candidate generation by structure guided method. The structure guided method is adapted from the TMG approach used for enumerating subtrees in our previous work [8]. Experiments utilizing very large databases of sequences which compare our technique with the existing technique, PLWAP [4], demonstrate the effectiveness of our proposed technique.
Keywords: mining frequent subsequences, sequence, phylogenic tree, sequential strips.

1 Introduction

Data mining strives to find frequent patterns in data [1–4, 12]. A task in data mining is characterized by the type of data, structures and patterns to be found [8, 9]. Advancements in data collection technologies have contributed to the high volumes of sales data. Sales data typically consists of a transaction timestamp and the list of items bought by customers. If one is interested in inter-transactional patterns, sales data is a good example of a sequential pattern. Other

WIT Transactions on Information and Communication Technologies, Vol 37, © 2006 WIT Press
www.witpress.com, ISSN 1743-3517 (on-line)
doi:10.2495/DATA060321

examples of sequential patterns include DNA sequences in DNA databases and events that are ordered in time in real time databases.

There are two different sub problems in mining frequent subsequences, i.e. problems of addressing (1) sequences of items and (2) sequences of itemsets. Addressing sequences of itemsets, i.e. sequences whose elements consist of multi items, is a more difficult problem and poses more challenges. In this paper, we will limit our scope of study to the first problem by comparing our technique, SEQUEST, with the PLWAP algorithm [4]. Nonetheless, SEQUEST using DMA-Strips is designed to tackle both sub problems of mining frequent subsequences.

Our contributions are as follows. We develop a DMA-Strips that provides a direct memory access to sequences in the database. One way to achieve efficient processing is to develop an intermediate memory construct that helps us to easily process and manipulate something. We showed this in [9] by developing an efficient intermediate memory construct called Embedding List (EL) [9, 10] to efficiently mine frequent embedded subtrees. The problem of using such intermediate memory constructs is that it demands some additional memory storage. For mining a very large database of sequences this is more a trade off than an advantage. DMA-Strips does not store hyperlinks of items, instead it segmented the database of sequences systematically so that it allows a direct access processing and manipulation. This contributes to a better speed and space performance than using an intermediate memory construct like EL that reserves extra storage in memory to store a corpus of hyperlinks of items. The idea of using hyperlinks is described in [9, 10]. Nevertheless, DMA-Strips resembles EL in that it allows efficient enumeration using a model or structure guided approach. Previously we developed the Tree Model Guided (TMG) [9, 10] candidate generation technique that utilizes the tree model to generate candidate subtrees. A unique property of the structure or model guided approach is that it ensures all enumerated patterns are valid patterns. These patterns are valid in the sense that they exist in the database, and hence by ensuring only valid patterns, no extra work is required to prune invalid patterns. The strategy to tackle mining sequence of items and itemsets using DMA-Strips is outlined in detail in chapter 2. Additionally for efficient frequency counting we use a hybrid method. For counting frequent 1-subsequences we transpose the database from horizontal format to vertical format [12]. We combine an efficient horizontal counting using a hashtable [3, 8 ,9] for counting 2-subsequences and the space efficient vertical join counting approach [3, 12] for counting k-subsequences where $k \geq 3$. This hybrid approach overcomes a non-linear performance for counting 2-subsequences due to the $O(n^2)$ complexity of the join enumeration approach [3, 8]. The rest of the paper is organized as follows. Section 2 gives the problem decomposition. Section 3 discusses related works. We examine the proposed technique and present the algorithms in section 4. In section 5, we empirically evaluate the performance and scale-up properties of the proposed technique. We summarize the research in section 6.

2 Problem statements

A sequence S consists of ordered elements $\{e_1,\ldots, e_n\}$. Sometimes elements of a sequence are referred to as events [12]. Each element in a sequence can be either an atomic or a complex type. For a sequence with atomic elements there is one entity or item per element. If the element is of a complex type, each element can have multiple items (itemsets). The itemsets can be ordered or unordered. In this paper we will show how our approach is designed to mine both *sequences of ordered itemsets* and *sequences of ordered items*. We refer to sequences of ordered itemsets or items simply as sequences of itemsets of items.

A sequence δ is denoted as $e_1 \rightarrow e_2 \rightarrow \ldots \rightarrow e_n$ and $|e_i|$ refers to the size of itemsets. We refer to the first item of the sequence as its *root*. In case of sequences of itemsets, the root is the first item of the first itemset in the sequence. Zaki [12] described $e_1 \rightarrow e_2 \rightarrow \ldots \rightarrow e_n$ to refer to the same notation. A sequence can be identified uniquely through its encoding which is a string of element labels separated by an arrow →. In the case of a sequence of items we assume each event e_i as an atomic item. Encoding of a sequence α is denoted by φ(α). An element with *j* items is denoted by $e{:}\{\lambda_1,\ldots, \lambda_j\}$ where λ_i is a label of item *i*. A label of an element with *j* items, denoted by λ(e), is a concatenation of $\lambda_1+\ldots+\lambda_j$. Say a sequence $\alpha{:}\{e_1,e_2\}$ where $e_1{:}\{A,B\}$ and $e_2{:}\{C,D\}$, the label of e_1, $\lambda(e_1)$, is 'AB' and the label of element e_2, $\lambda(e_2)$, is 'CD' and the encoding of sequence α, φ(α), is 'AB→CD'. This labelling function translates equivalently for an element that consists of an atomic item. We normally refer to a sequence by its encoding without quotes. Consider a sequence with n elements. We refer to a sequence with k items as k-sequence, i.e. *k* equals to the sums of $|e_i|$ for i=1,…,*n*. For example, AB→CD is a 4-sequence and AB→DF→G is a 5-sequence. A subsequence is a subset of a sequence such that the order of elements in the subsequence is the same as the order of the elements in the sequence. Accordingly, a subsequence with k elements is called a k-subsequence. We say that a subsequence S is a subset of sequence T if the elements of subsequence S are a subset of elements of sequence T and the ordering of items is preserved ($S \leq T$). A→C is an example of subsequence of AB→CD whereas AC is not. A is a subset of AB and C is a subset of CD whereas AC is not a subset of either AB or CD. We denote that element α is a subset of element β as $\alpha \subseteq \beta$. Further the ordering of elements in a sequence follows a certain ordering function. The most common ordering function is a time function. If an element α occurs at t_1 and element β occurs at t_2 where $t_1 < t_2$, the position of α and β in a sequence is *i* and *j* respectively and $i < j$. A *gap constraint* [7] is very similar to the notion of level of embedding [9] between two nodes in a tree with an ancestor-descendant relationship. The gap between two elements in a sequence is determined by the *gap distance* (Δ) between two elements at position *i* and *j*, i.e. Δ is a delta between *i* and *j*. A subsequence τ is said to be contiguous if there is no gap between every two consecutive elements. In other words, for a *contiguous subsequence* every two consecutive elements have a gap distance 1. Subsequences that contain any two consecutive elements with $\Delta > 1$ are called *non-contiguous subsequence*. Suppose we have a sequence

δ:A→BCD→F→GH→IJ→JK. A→B→F and A→F→K are both subsequences of δ. A→B→F is a contiguous subsequence of δ since the gap distance of every two consecutive elements is 1, whereas A→F→K is a non-contiguous subsequence of δ. Because the gap distance between A and F in A→F→K is 2 and between F and K is 3. The database D for sequence mining consists of input-sequences. Each input-sequence can contain one to many itemsets. Each input-sequence is identified through a unique sequence-id (sid) whereas each element (itemsets) is identified through a unique event-id (eid). If D contains sales data, the sid can be a customer-id. A common event identifier in transactional databases is a timestamp. We say that a sequence α has a support count γ if there are γ numbers of input-sequences that support it. An input-sequence supports sequence α if it contains at least 1 occurrence of α. Mining frequent subsequences from a database of input-sequences can be formulated as discovering all frequent subsequences whose support count γ is greater or equal to the user specified minimum support threshold σ.

3 Related works

Early in their work, Agrawal [1] introduced the problem of mining sequential patterns over transactions data. In [1] they proposed 2 algorithms for mining sequential data, AprioriAll and AprioriSome. While the problem of market basket analysis is concerned with finding frequent intra-transaction patterns, mining sequential patterns on the other hand is concerned with finding frequent inter-transaction patterns [5]. Further, the data used in mining sequential patterns has an ordered notion. A sequential pattern consists of ordered events or elements. We are in particular interested in finding non-contiguous subsequences [7]. Mining non-contiguous subsequences can be very expensive when data has long patterns. A concept of a constraint can be introduced. In [7] they proposed an improved algorithm GSP that generalizes the scope of sequential pattern mining to include time constraints, sliding time windows, and user-defined taxonomy. Mannila and Toivonen [6] proposed an approach that addresses the problem of sequential mining where each element of the sequences consists of atomic item. Similarly [11] formulated the problem of mining frequent subsequences from genetic sequences where each element of the sequences consists of an atomic item. Recently, Ezeife and Lu [4] developed an efficient technique PLWAP based on the concept of WAPTree that mines frequent subsequences of items. PLWAP is reported to have desirable scalability properties. Additionally it was reported that it outperforms WAPTree and GSP. Others [7, 12] formulated the problem of mining frequent subsequences from database of transaction sequences such that each element can comprise of multiple items. Both [7, 12] assume unordered itemsets. Zaki [12] developed an efficient algorithm, SPADE, for mining frequent subsequences. SPADE uses a vertical id-list database format and a lattice-theoretic approach to decompose the original search space into smaller pieces.

4 SEQUEST algorithm

4.1 Database scanning

SEQUEST scans the database S_{db} twice and generates a frequent database that contains only frequent 1-subsequences, S_{db}' after the first scan (figure 3 [1]). S_{db}' is obtained by intersecting S_{db} with L_1 (figure 3 [2]), the set of frequent 1-subsequences. The first scan and the second scan are executed in two separate runs. S_{db}' is used to construct DMA-Strips.

4.2 Constructing DMA-Strips

A DMA-Strip is constructed as follows (figure 3 [3]). For each item in S_{db}', an ordered list (strip) is generated which stores a sequence of items' *label*, *scope* and *eid*. Each strip is uniquely identified by a sequence-id *sid*. To allow mining sequences of itemsets each item in a strip stores event-id *eid* and a notion of *scope*. The *eid* groups items in a strip based on their timestamps. The *scope* is used to determine the relationship between two consecutive items in a strip. The *scope* of an item A is determined by the position of the last item that belongs to the same event. If the preceding item's position is out of the prior item's scope this tells us for sequence of itemsets that the two items (say A and B) occur at different times, i.e. A→B. Otherwise A and B occurs at the same time the event is generated, i.e. AB. Through the use of the *scope* DMA-Strips can be used for both mining sequences of itemsets or items.

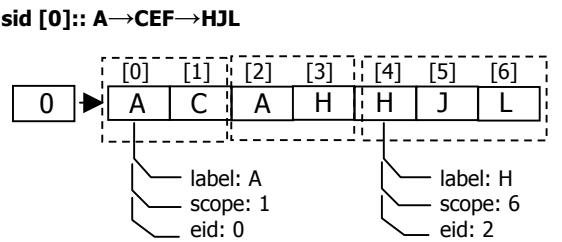


Figure 1: Pseudo-code of SEQUEST.

Figure 1 shows an example of a strip of a sequence A→CEF→HJL whose sid is 0. The above sequence consists of 3 different events. AC, EF, and HJL belong to event 0, 1 and 2 respectively. The scope of A is equal to the position of the last item whose eid is the same as A, which is C. Hence, A's scope is determined to be 1. Similarly, using the same rule, H's scope is determined to be 6. Each strip can be fetched in O(1) by specifying its offset or its sid.

4.3 Enumeration of subsequences

The enumeration of subsequences is done by iterating a strip and extending one item at the time to the expanding k-subsequence. We call this a model-guided or

structured-guided enumeration as the enumeration is guided by the actual model or structure of the data. Figure 2 illustrates how the enumeration of AC, A→E, and A→H is done through the strip. Given that we have 1-subsequence *I* at position *r* and the strip *S* with size |*S*|, 2-subsequences are generated by extending *I* with each item *J* at position *t* on its right to the end of the strip such that $r < t \leq |S|$ (figure 3 [4]). This can be generalized to enumerating (k+1)-subsequences from a k-subsequence (figure 3 [5]). For a k-subsequence it is sufficient to store only a pair of *sid* and the last item's position (*tail*) as its occurrence coordinate. Hence, given an occurrence coordinate of a (k-1)-subsequence its occurrence coordinate will be in a form of (*sid*,*tail*). Hence the enumeration of (k+1)-subsequences M using a strip *S* from a k-subsequence *s* whose encoding is $\varphi(s)$, and occurrence coordinate is (p,q), where *p* is its *sid* and q is the *tail* can be formalized as follows. Suppose that C is the set of occurrence coordinates of (k+1)-subsequences, $C=\{(p,t)|q< t \leq |S|\}$. The encoding of the (k+1)-subsequences is computed by performing a scope test. We define two functions, $\Psi(p,q)$ and $\lambda(p,q)$ that determine the scope and label of an item in a strip with *sid p* and at position *q*. If $t > \Psi(p,q)$ then the new encoding $\varsigma = \varphi(s)$ + '→' + $\lambda(p,t)$ otherwise $\varsigma = \varphi + \lambda(p,t)$ (figure 3 [6]and[7]). For example, from figure 2, a strip with *sid* 0, extending A_0 (A at position 0) with A_2 (A at position 2) generates A→A since $2 > \Psi(0,0)$ for the given strip. We know that $\Psi(0,0)$ from figure 1 is 1. Although a constraint represents an important concept, we don't particularly focus on it in this paper. However the current scheme can be extended easily to incorporate the *gap constraint* described in the previous section for example by restricting the extension using the event-id delta.

4.4 Frequency counting

We utilize a hybrid strategy enumerating subsequences by extension and counting by the vertical join approach to make use of the desired property from each technique. Enumeration by extension makes sure that the candidates generated are all valid. What we mean by a valid candidate is a candidate whose support is greater than zero. The vertical counting approach is used to help us achieve a space efficient and less expensive full pruning strategy.

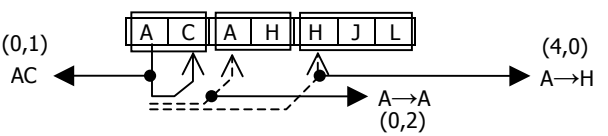


Figure 2: Enumeration examples.

We define two operators *inter-join* and *intra-join* (figure 3 [10, 11]) to tackle both sub problems of mining sequence of itemsets and items. The inter-join operator is used whenever the tail items of two joinable (k-1)-subsequences are from different events, whereas *intra-join* operator is used if they are from the same event. *Intra-join* will join any two occurrence coordinates whose *sid* and *eid* are equal. Inter-join will join any two occurrence coordinates whose *sid* are equal but the prior *eid* is less than the preceding *eid*. If the *gap constraint* is to be

introduced, it will be to define the *inter-join* operator by constraining the gap between the two eid. Space efficiency is achieved through the vertical join approach by freeing the memory sooner than the horizontal counting approach does. The details of how the less expensive full pruning is achieved through the vertical join approach is discussed below.

```
Sdb:Sequence Database
Lk:Frequent k-subsequences

L1= Scan-Sdb                                                    [1]
Sdb'= Intersect(Sdb∩L1)                                         [2]
DMA-STRIPS = Construct-DMA-STRIPS(Sdb')                         [3]
L2 = Generate-2-Subsequences(DMA-STRIPS)                        [4]
k=3
while(Lk > 0)
   Lk= Generate-k-Subsequences(Lk-1)                            [5]
   k++
Generate-k-Subsequences(Lk-1):
for each (k-1)-sequence s in Lk-1{
  for each occurrence-coordinate oc(sid,r) in s{
    for t=r+1 to |DMA-STRIPS[sid]|{
       if( t > scope(sid,r) )
           join = inter-join
           φ(s') = φ(s) + → + label(sid,t)                      [6]
       else
           join = intra-join
           φ(s') = φ(s) + label(sid,t)                          [7]
       root-pruned = remove-root(φ(s'))                         [8]
       tail-pruned = φ(s')                                      [9]
       if(join == intra-join)
          Intra-Join(root-pruned, tail-pruned)                  [10]
       else
          Inter-Join(root-pruned, tail-pruned)                  [11]
       if( InFrequent(φ(s')) )
          Prune(φ(s'))
       else
          Insert(φ(s'),Lk)
```

Figure 3: Pseudo-code of SEQUEST.

4.5 Pruning

According to Apriori theory a pattern is frequent if and only if all its subsets are frequent. To make sure that all generated subsequences do not contain infrequent subsequences, full (k-1) pruning must be performed. Full (k-1) pruning or *full pruning* implies that at most (k-1) numbers of (k-1)-subsequences need to be generated from the currently expanding k-subsequences. This process is expensive. Using the join approach principle, any two (k-1)-subsequence that are obtained from a k-subsequence by removing one node at a time can be joined to form the k-subsequence back. This implies that we could accelerate the full pruning by only doing root and tail pruning (figure 3 [8, 9]). Root pruning is

done by removing the root node, i.e. the first node of the sequence and checking if the pattern has been generated previously. Similarly, tail pruning is done by removing the tail node. Since we use a hashtable to perform frequency counting this can be done by simply checking if the pattern exists in the k-1 hashtable. If it does not exist we can safely prune the generated k-subsequences. However, it is not completely safe to assume that if both root-pruned and tail-pruned (k-1)-subsequences exist means the generated k-subsequences must be frequent. At least, the root and tail pruning will do the partial check and the final check is done by joining the occurrence coordinates of the root-pruned and tail-pruned (k-1)-subsequences which is done as part of the support counting. This scheme can only be done using vertical support counting since vertical support counting counts the support of a subsequence vertically, i.e. the result is known immediately. Horizontal counting, on the other hand, increments the support of a subsequence by one at a time and the final support of a subsequence is not known until the database is traversed completely.

5 Experimental results and discussions

This section explores the effectiveness of the proposed SEQUEST algorithm, by comparing it with the PLWAP [4] algorithm that processes sequences of items. The formatting of the datasets used by PLWAP can be found in [4]. SEQUEST, on the other hand, uses the datasets format that is used by the IBM data generator [1, 7]. The minimum support σ is denoted as (Sxx), where xx is shown as the absolute support count. Experiments were run on Dell Rack Optimized Slim Servers (dual 2.8 Ghz CPU, 4Gb RAM) running Fedora 4.0 ES Linux. The source code for each algorithm was compiled using GNU g++ version 3.4.3 with –g and –O3 parameters. The datasets chosen for this experiment were previously used in [4], in particular data.ncust_25 (figure 4), data.100K, data.600K, data.ncust_125 (figure 5 and 6(a)), and data.ncust_1000 (figure 6(b)). To obtain the 500K dataset for experiments in figure 5 we cut the last 100K sequences from the data.600K dataset.

Figure 4(a) shows the comparison of the algorithms with respect to time, for varying support thresholds. In this experiment we try two schemes for SEQUEST, namely SH (refers to SEQUEST-hybrid) and S (refers to SEQUEST). SH uses a vertical join counting approach only for generating frequent k-sequences where k>3, whereas S uses a horizontal counting approach using a hashtable for all k. The results in figure 4(a) show that SH performs better than S and PLWAP. The performance of SH remains nearly constant for a very small support (σ:2 or 0.001%). The difference between SH and PLWAP performance at σ:2 is in the order of ~800 times. It should be noted as well that the memory usage of SH is much better than S due to the vertical counting approach used in SH.

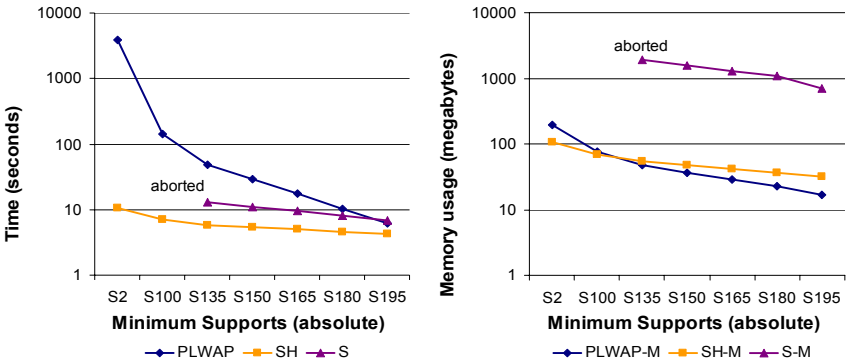


Figure 4: 200K dataset (a) time performance (b) memory profiles.

Figure 4(b) shows comparison of memory usage between PLWAP, SH and S. We can see that when the horizontal counting approach is used S suffers from memory blow-up and this contributes to S aborting at $\sigma \leq 135$. SH memory profiles on the other hand, show a superior overall performance over the other two, especially when the minimum support is lowered.

We perform a scalability test by varying the datasets of different sizes while σ is fixed at 0.005%. In figure 5(a), SH outperformed PLWAP with respect to time in the order of ~250 times faster. In this experiment we try two different methods for generating frequent 2-subsequences. SH uses the structure-guided enumeration approach as described in the previous section, and S-L2J uses a join approach. Figure 6(a) shows that SH has a more linear performance than S-L2J. We observe that if the enumeration for 2-subsequences uses the join approach the time performance is up by roughly n^2 if the datasets size is scaled up by n times. On the other hand, it is interesting to see that whenever the join approach is used the memory usage is slashed by almost half.

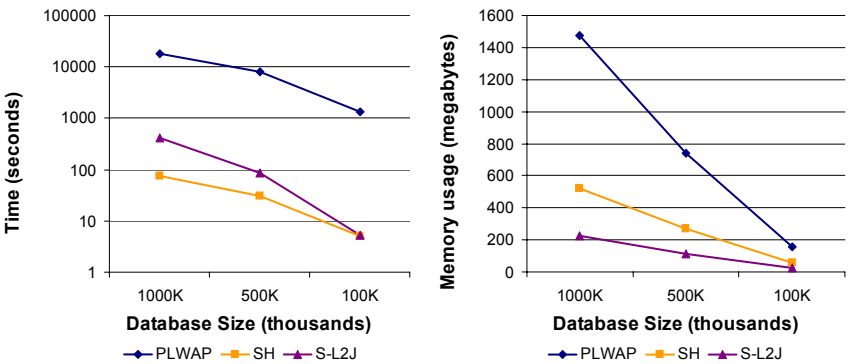
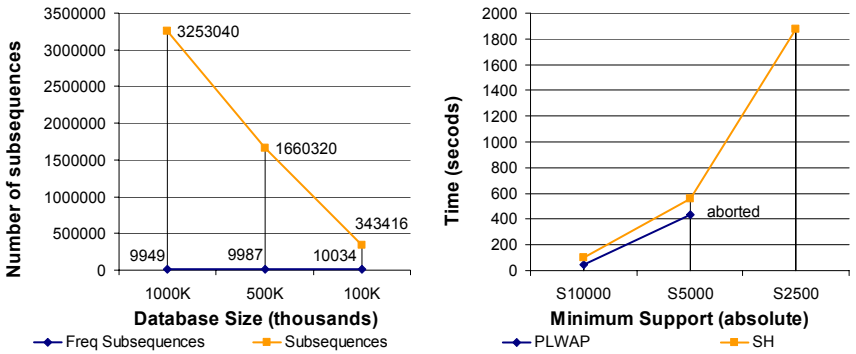


Figure 5: 1000K dataset (a) time performance (b) memory profiles.



(a) (b)

Figure 6: (a) 1000K dataset Frequency distribution (b) 7000K time performance.

Figure 6(b) shows that SH memory usage is 2x greater than S-L2J. Overall, we witness that all SEQUEST variants, SH and S-L2J have outperformed PLWAP in terms of speed and space performance. Figure 6(a) shows the distribution of candidate subsequences and frequent subsequences generated over different dataset sizes with σ fixed at 0.005%. Figure 6(b) shows the time performance comparison between PLWAP and SH for a dataset of 7.8 million sequences. For this particular dataset, we see that PLWAP outperforms SH by a small margin. However at σ:2500 (0.03%), PLWAP aborted due to a memory blow up problem. At σ:10000 and σ:5000 the number of extracted frequent subsequences is relatively low, 130 and 1211 respectively.

PLWAP performs well whenever the average length of the extracted subsequences is relatively short. In other words, PLWAP performs well for large sparse databases but when applied to dense databases with long patterns, the performance is significantly reduced. In figure 4 and 5 experiments we have shown that PLWAP performance degrades whenever the support is set at a lower value and the number of frequent patterns extracted is very high. On the other hand, we observe that SH performs well for extracting frequent 1 and 2 subsequences. However the performance started to degrade when generating 3-subsequences. For this very large dataset we see that the bottleneck of SH has shifted from k=2 to k=3. We observe that the number of candidates generated for k=3 is extremely large, i.e. 10,137,400. From previous experiment in figure 5(a) we saw a similar property, where counting using the vertical join approach when the number of candidates are very large at k=2, S-L2J, although space efficient, suffered a non-linear performance. From this we can infer that optimal performance can be obtained whenever it is known when to switch from the horizontal to vertical join counting approach. The horizontal counting approach suffers from memory blow-up but is faster whenever the numbers of candidates generated are very large. The vertical counting join approach is space efficient however the performance degrades when the number of candidates to be generated is very large.

6 Concluding remarks

Overall remarks about the algorithms are as follows. PLWAP is a space efficient technique but it suffers when the support is lowered and the numbers of frequent subsequences extracted are high. SEQUEST, on the other hand, when the horizontal counting approach is used suffers from a memory blow-up problem due to the BFS approach for generating candidate subsequences. It uses two hash-tables and additional space needs to be occupied when the length of the sequences to be enumerated increases. We also show that if SEQUEST uses a vertical join counting approach it performs extremely well for both speed and space performance. If the vertical join counting approach is used, the space can be freed much sooner than if the horizontal counting approach is used. The hybrid method of structure-guided enumeration using DMA-Strips and the vertical join counting approach enables SEQUEST to tackle a very large database and shows a linear scalability performance. However it should be noted that whenever the frequency of extracted subsequences are high the vertical join approach performance could degrade due to the nature of the join approach complexity. Additionally, through using the notion of *scope* in DMA-Strips, SEQUEST can process sequences of itemsets as well as sequences of items.

References

[1] Agrawal, R., Srikant, R.: Mining Sequential Patterns. In Proc. IEEE 11th ICDE (1995) Vol. 6, No. 10, 3–14

[2] Bayardo, R. J., Efficiently Mining Long Patterns from Databases. SIGMOD'98, Seattle, WA, USA (1998)

[3] Chi, Y., Nijssen, S., Muntz, R.R., Kok. J.N.: Frequent Subtree Mining An Overview. Fundamenta Informaticae, Special Issue on Graph & Tree Mining (2005)

[4] Ezeife, C.I., Lu. Y.: Mining Web Log sequential Patterns with Position Coded Pre-Order Linked WAP-tree. The International Journal of Data Mining and Knowledge Discovery (DMKD) (2005) Vol. 10, 5-38

[5] Feng, L., Dillon, T.S., and Liu, J.: Inter-transactional association rules for multi-dimensional contexts for prediction and their application to studying meteorological data. Data Knowl. Eng. (2001) Vol. 37, No. 1, 85-115.

[6] Manilla, H., Toivonen, H.: Discovering generalized episodes using minimal occurrences. In 2nd Intl. Conf. Knowledge Discovery and Data Mining (1996)

[7] Srikant, R., Agrawal, R.: Mining Sequential Patterns: Generalizations and Performance Improvements. Technical Reports, IBM Research Division, Almaden Research Centre, San Jose, CA (1996)

[8] Tan, H., Dillon, T.S., Feng, L., Chang, E., Hadzic, F.: X3-Miner: Mining Patterns from XML Database. In Proc. of Data Mining '05. Skiathos, Greece (2005)

[9] Tan, H., Dillon, T.S., Hadzic, F., Feng, L., Chang, E.: IMB3-Miner: Mining Induced/Embedded Subtrees by Constraining the Level of Embedding. In Proc. of PAKDD'06 (2006) Singapore
[10] Tan, H., Dillon, T.S., Hadzic, F., Feng, L., Chang, E.: Tree Model Guided Candidate Generation for Mining Frequent Patterns from XML Documents. ACS TOIS Journal (2005) (Submitted)
[11] Wang, J.T-L, Chirn, G.-W, Marr, T.G., Shapiro, B., Shasha, D., Zhang, K. Combinatorial pattern discovery for scientific data: Some preliminary results. In Proc. of the ACM SIGMOD Conference on Management of Data, Minneapolis, May 1994.
[12] Zaki, M.J.: SPADE: An Efficient Algorithm for Mining Frequent Sequences. Machine Learning (2000) Vol. 0, 1-31