Under consideration for publication in Knowledge and Information Systems

TSP: Mining Top-K Closed Sequential Patterns

Petre Tzvetkov, Xifeng Yan and Jiawei Han

Department of Computer Science, University of Illinois at Urbana-Champaign, Illinois, USA

Abstract. Sequential pattern mining has been studied extensively in data mining community. Most previous studies require the specification of a *min_support* threshold for mining a complete set of sequential patterns satisfying the threshold. However, in practice, it is difficult for users to provide an appropriate *min_support* threshold. To overcome this difficulty, we propose an alternative mining task: mining top-k frequent closed sequential patterns of length no less than min_ℓ, where k is the desired number of closed sequential patterns to be mined, and min_ℓ is the minimal length of each pattern. We mine the set of *closed patterns* since it is a compact representation of the complete set of frequent patterns.

An efficient algorithm, called TSP, is developed for mining such patterns without *min_support*. Starting at (absolute) *min_support* = 1, the algorithm makes use of the length constraint and the properties of top-k closed sequential patterns to perform dynamic support-raising and projected database-pruning. Our extensive performance study shows that TSP has high performance. In most cases, it outperforms the efficient closed sequential pattern mining algorithm, CloSpan, even when the latter is running with the best tuned *min_support* threshold. Thus we conclude that for sequential pattern mining, mining top-k frequent closed sequential patterns without *min_support* is more preferable than the traditional *min_support*-based mining.

1. Introduction

Sequential pattern mining is an important data mining task that has been studied extensively (Agrawal and Srikant, 1995) (Mannila et al, 1995) (Guha et al, 1999) (Pei et al, 2001) (Zaki, 2001) (Ayres et al, 2002). It has a broad range of applications, including analysis of customer purchase patterns, web access patterns, discovery of motifs and tandem repeats in DNA sequences, analysis of various

Received Nov 19, 2003
Revised Jan 19, 2004
Accepted Feb 16, 2004

sequencing or time-related processes such as scientific experiments, disease treatments, natural disasters, and many more.

The sequential pattern mining problem was first introduced by Agrawal and Srikant (Agrawal and Srikant, 1995): *Given a set of sequences, where each sequence consists of a list of elements and each element consists of a set of items, and given a user-specified min_support threshold, sequential pattern mining is to find all of the frequent subsequences whose occurrence frequency is no less than min_support.*

The common framework among the current sequential pattern mining methods is to use a *min_support* threshold to generate the frequent sequential patterns, based on the popular Apriori property (Agrawal and Srikant, 1994): *every subpattern of a frequent pattern must be frequent* (also called the *downward closure property*). This framework leads to the following two problems that may hinder its popular use.

First, sequential pattern mining often generates an exponential number of patterns, which is unavoidable when the database consists of long frequent sequences. The similar fact is observed at mining itemset and graph patterns when the size of the patterns is large. For example, a database containing a frequent sequence $\langle(a_1)(a_2)\ldots(a_{64})\rangle$ $(\forall i \neq j, a_i \neq a_j)$ will generate at least $2^{64}-1$ frequent subsequences. It is very likely some subsequences share the same support with this long sequence, and they are essentially redundant patterns.

Second, it is nontrivial to provide an appropriate min_support threshold: one needs to have prior knowledge about the mining query and the task-specific data, and be able to estimate, without mining, how many patterns will be generated with a particular threshold. Setting *min_support* is a subtle task: *a too small value may lead to the generation of thousands of patterns, whereas a too big one may lead to no answers found.*

A solution to the first problem, called CloSpan, was proposed recently (Yan et al, 2003). CloSpan can mine closed sequential patterns, where a sequential pattern s is closed if there exists no superpattern of s with the same support in the database. Mining closed patterns may significantly reduce the number of patterns generated and is *information lossless* because it can be used to derive the complete set of sequential patterns.

As to the second problem, the similar issue also occurs in frequent itemset mining. Han et al. (Han et al, 2002) changes the task of mining frequent patterns to *mining top-k frequent closed patterns of minimal length min_ℓ*, where k is the number of closed patterns to be mined, top-k refers to the k most frequent patterns, and *min_ℓ* is the minimal length of the closed patterns. This setting is also desirable in the context of sequential pattern mining. We will show a real application case of top-k sequential pattern mining in Section 2. Unfortunately, most of the techniques developed in (Han et al, 2002) cannot be directly applied in sequence mining. This is because subsequence testing requires order matching which is more difficult than subset testing. Moreover, the search space of sequences is much larger than that of itemsets. However, some ideas developed in (Han et al, 2002) are still influential in our algorithm design.

In this paper, we introduce a new multi-pass search space traversal algorithm that finds the most frequent patterns early in the mining process and allows dynamic raising of the *min_support* threshold which is then used to prune unpromising branches in the search space. Also, we propose an efficient closed pattern verification method which guarantees that during the mining process the candidate result set consists of desired closed sequential patterns. The efficiency

of our mining algorithm is further improved by applying the minimum length constraint in the mining and by employing the early termination conditions developed in CloSpan (Yan et al, 2003).

The performance study shows that in most cases our algorithm TSP has comparable or better performance than CloSpan, currently the most efficient algorithm for mining closed sequential patterns, even when CloSpan is running with the best tuned *min_support*.

The rest of the paper is organized as follows. In Section 2, some basic concepts of sequential pattern mining are introduced and the problem of mining the top-k frequent sequential patterns without minimum support is formally defined. Section 3 presents the algorithm for mining top-k frequent closed sequential patterns. A performance study is reported in Section 4. Section 5 gives an overview of the related work on sequential pattern mining and top-k frequent patten mining. We also discuss extensions of our method and suggestion for future research in this section. Section 6 concludes this study.

2. Problem Definition

In this section we define the basic concepts in sequential pattern mining and introduce the problem of mining the top-k frequent sequential patterns. The notations used here are similar to (Yan et al, 2003).

Let $I = \{i_1, i_2, \ldots, i_k\}$ be a set of all items. A subset of I is called an *itemset*. A *sequence* $s = \langle t_1, t_2, \ldots, t_m \rangle$ ($t_i \subseteq I$) is an ordered list. Without loss of generality, we assume that the items in each itemset are sorted in certain order (such as alphabetic order). The *size*, $|s|$, of a sequence is the number of itemsets in the sequence, i.e, $|s| = m$. The *length*, $l(s)$, is the total number of items in the sequence, i.e., $l(s) = \sum_{i=1}^{m} |t_i|$. A sequence $\alpha = \langle a_1, a_2, \ldots, a_m \rangle$ is a *sub-sequence* of another sequence $\beta = \langle b_1, b_2, \ldots, b_n \rangle$, denoted as $\alpha \sqsubseteq \beta$ (if $\alpha \neq \beta$, written as $\alpha \sqsubset \beta$), if and only if $\exists i_1, i_2, \ldots, i_m$, such that $1 \leqslant i_1 < i_2 < \ldots < i_m \leqslant n$ and $a_1 \subseteq b_{i_1}, a_2 \subseteq b_{i_2}, \ldots,$ *and* $a_m \subseteq b_{i_m}$. We also call β a *super-sequence* of α, and β *contains* α. If β *contains* α and their supports are the same, we call β *absorbs* α.

A sequence database, $D = \{s_1, s_2, \ldots, s_n\}$, is a set of sequences. Each sequence is associated with an *id*. For simplicity, say the *id* of s_i is i. $|D|$ represents the number of sequences in the database D. The (absolute) *support* of a sequence α in a sequence database D is the number of sequences in D which contain α, $support(\alpha) = |\{s | s \in D \text{ and } \alpha \sqsubseteq s\}|$.

Definition 2.1. (top-k closed sequential pattern) A sequence s is a **sequential pattern** in a sequence database D if its support (i.e., occurrence frequency) in D is no less than *min_support*. A sequential pattern s is a **closed sequential pattern** if there exists no sequential pattern s' such that (1) $s \sqsubset s'$, and (2) $support(s) = support(s')$. A closed sequential pattern s is a **top-k closed sequential pattern of minimal length** min_ℓ if there exist[1] no more than $(k-1)$ closed sequential patterns whose length is at least min_ℓ and whose support is higher than that of s. ■

[1] Since there could be more than one sequential pattern having the same support in a sequence database, to ensure the result set is independent of the ordering of transactions, the proposed method will mine every closed sequential pattern whose support is no less than the support of the k-th frequent closed sequential pattern.

Seq ID.	Sequence
0	$\langle(ac)(d)(e)\rangle$
1	$\langle(e)(abcf)(e)\rangle$
2	$\langle(a)(e)(b)\rangle$
3	$\langle(d)(ac)(e)\rangle$

Table 1. Sample Sequence Database D

Our task is to *mine the top-k closed sequential patterns of minimal length min_ℓ efficiently in a sequence database.*

Example 1. Table 1 shows a sample sequence database. We refer to this dataset as D and will use it as a running example in the paper. Suppose our task is to find the top-2 closed sequential patterns with $min\_\ell = 2$ in D. The output should be: $\langle(a)(e)\rangle : 4, \langle(ac)(e)\rangle : 3$. Although there are two more patterns with support equal to 3: $\langle\langle(ac)\rangle : 3, \langle(c)(e)\rangle : 3$, they are not in the result set because they are not closed and both of them are absorbed by $\langle(ac)(e)\rangle : 3$. ■

Application scenario. Although top-k sequential pattern mining has its applications in customer shopping sequence mining, it is interesting to note that it can be applied to improve the performance of computer storage systems (Li et al, 2003). Li et al. (Li et al, 2003) applies CloSpan to find block correlations in disk access sequences. A disk access sequence is a sequence of blocks like $b_{35}, b_{100}, b_{9039}, ...$; b_i represents the i_{th} block on the disk. Suppose an access to b_{35} is repeatedly followed by an access to b_{9039}, it may improve the I/O performance if we arrange these two blocks adjacent or fetch them together. When we mine closed sequential patterns in disk access sequences, the number of sequential patterns returned may vary a lot based on different support thresholds. In practice, it is difficult for users to provide an appropriate support threshold. However, the users may have an estimation about the number of patterns they are able to process, especially, in system caching and prefetching. In most cases, it is sufficient to achieve good performance by optimizing the top thousands of correlated blocks. Thus top-k sequential pattern mining paves the way for this kind of application.

3. Method Development

This section presents our method, TSP, for mining top-k closed sequential patterns without a given minimum support threshold. First, we introduce the concept of prefix projection-based sequential pattern mining and the PrefixSpan algorithm (Pei et al, 2001) which provides the background for the development of our method. Next, we present a novel multi-pass search space traversal algorithm for mining the most frequent patterns and an efficient method for closed pattern verification and the minimum support raising during the mining process. Finally, we propose two additional optimization techniques that further improve the efficiency of the algorithm.

3.1. Projection-based Sequential Pattern Mining

Here we briefly introduce PrefixSpan (Pei et al, 2001) and CloSpan (Yan et al, 2003), and then focus on the design of TSP.

Definition 3.1. Given a sequence $s = \langle t_1, \ldots, t_m \rangle$ and an item α, $s \diamond \alpha$ means s concatenates with α. $s \diamond \alpha$ can be an **I-Step extension** (Ayres et al, 2002), $s \diamond_i \alpha = \langle t_1, \ldots, t_m \cup \{\alpha\} \rangle$, if $\forall k \in t_m, k < \alpha$; or an **S-Step extension** (Ayres et al, 2002), $s \diamond_s \alpha = \langle t_1, \ldots, t_m, \{\alpha\} \rangle$. ■

For example, $\langle (ae) \rangle$ is an I-Step extension of $\langle (a) \rangle$, whereas $\langle (a)(c) \rangle$ is an S-Step extension of $\langle (a) \rangle$. (Yan et al, 2003) extends the definition of item extension to sequence extension.

Definition 3.2. Given two sequences, $s = \langle t_1, \ldots, t_m \rangle$ and $p = \langle t'_1, \ldots, t'_n \rangle$, $s \diamond p$ means s concatenates with p. It can be **itemset-extension**, $s \diamond_i p = \langle t_1, \ldots, t_m \cup t'_1, \ldots, t'_n \rangle$ if $\forall k \in t_m, j \in t'_1, k < j$; or **sequence-extension**, $s \diamond_s p = \langle t_1, \ldots, t_m, t'_1, \ldots, t'_n \rangle$. If $s' = p \diamond s$, p is a **prefix** of s' and s is a **suffix** of s'. ■

For example, $\langle (ac) \rangle$ is a prefix of $\langle (ac)(d)(e) \rangle$ and $\langle (d)(e) \rangle$ is its suffix.

Definition 3.3. An **s-projected** database is defined as $D_s = \{p \mid s' \in D, s' = r \diamond p$ such that r is the minimum prefix (of s') containing s (i.e., $s \sqsubseteq r$ *and* $\nexists r', s \sqsubseteq r' \sqsubset r)\}$. Notice that p can be empty. ■

For Table 1, $D_{\langle (ac) \rangle} = \{\langle (d)(e) \rangle, \langle (\_f)(e) \rangle, \langle (e) \rangle\}$, where$(_f)$ means that f and item c in $\langle (ac) \rangle$ come from the same itemset. For each suffix sequence p in D_s, the type of extension, i.e., whether s' is an itemset-extension or a sequence-extension of s, is recorded. The type of extension helps correctly grow s using the projected database.

Assume that there exists a lexicographic order in the set of all items in a database. *Set Lexicographic Order* is a linear order defined as follows. Let $t = \{i_1, i_2, \ldots, i_k\}, t' = \{j_1, j_2, \ldots, j_l\}, where\ i_1 \leqslant i_2 \leqslant \ldots \leqslant i_k\ and\ j_1 \leqslant j_2 \leqslant \ldots \leqslant j_l$. Then $t < t'$ iff either of the following is true:

1. for some h, $1 \leqslant h \leqslant min\{k, l\}$, we have $i_r = j_r$ for $r < h$, and $i_h < j_h$, or
2. $k < l$, and $i_1 = j_1$, $i_2 = j_2, \ldots, i_k = j_k$.

For example, $(a, f) < (b, f)$, $(a, b) < (a, b, c)$, and $(a, b, c) < (b, c)$.

Based on this set lexicographic order, *Sequence Lexicographic Order* is given as follows: (i) if $s' = s \diamond p$, then $s < s'$; (ii) if $s = \alpha \diamond_i p$ and $s' = \alpha \diamond_s p'$, no matter what the order relation between p and p' is, $s < s'$; (iii) if $s = \alpha \diamond_i p$ and $s' = \alpha \diamond_i p'$, $p < p'$ indicates $s < s'$; and (iv) if $s = \alpha \diamond_s p$ and $s' = \alpha \diamond_s p'$, $p < p'$ indicates $s < s'$.

For example, $\langle (ab) \rangle < \langle (ab)(a) \rangle$ (i.e., a sequence is greater than its prefix); $\langle (ab) \rangle < \langle (a)(a) \rangle$ (i.e., a sequence-extended sequence is greater than an itemset-extended sequence if both of them share the same prefix).

We construct a Lexicographic Sequence Tree as follows:

1. each node in the tree corresponds to a sequence, and the root is a *null* sequence;
2. if a parent node corresponds to a sequence s, its child is either an itemset-extension of s, or a sequence-extension of s; and
3. the left sibling is less than the right sibling in sequence lexicographic order.

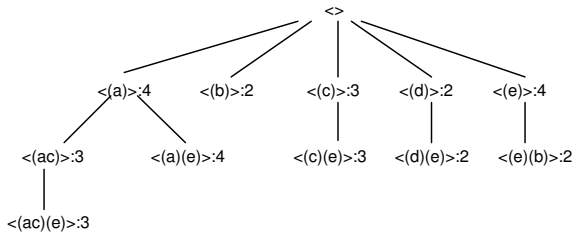


Fig. 1. Lexicographic Sequence Tree

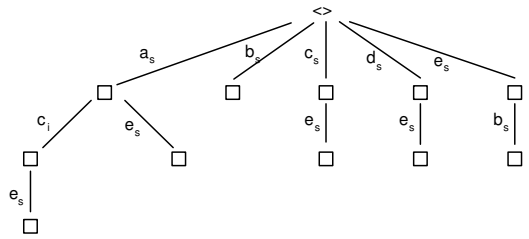


Fig. 2. Prefix Search Tree

Figure 1 shows a lexicographic sequence tree for mining the sample database in Table 1 with $min\_support = 2$. The numbers in the figure represent the support of each frequent sequence. We define the level of a node by the number of edges from the root to this node. If we do pre-order transversal in the tree, we can build an operational picture of lexicographic sequence tree (Figure 2). It shows that the process extends a sequence by adding an I-Step item or an S-Step item.

Algorithm 3.1 PrefixSpan

Input: A sequence s, a projected DB D_s, and $min\_support$.
Output: The frequent sequence set F.

1: insert s to F;
2: scan D_s once, find every frequent item α such that
   (a) s can be extended to $(s \diamond_i \alpha)$, or
   (b) s can be extended to $(s \diamond_s \alpha)$;
3: **if** no such α available **then**;
4: **return**;
5: **for each** α **do**
6: PrefixSpan($s \diamond_i \alpha$, $D_{s\diamond_i \alpha}$, $min\_support$, F); or
7: PrefixSpan($s \diamond_s \alpha$, $D_{s\diamond_s \alpha}$, $min\_support$, F);
8: **return**;

Algorithm 3.1 from PrefixSpan (Pei et al, 2001) provides a general framework for depth-first search in the prefix search tree. For each discovered sequence s and its projected database D_s, it performs I-Step extension (line 6) and S-

Step extension (line 7) recursively until all the frequent sequences which have the prefix s are discovered. Line 3 shows the termination condition: when the number of sequences in the s-projected database is less than $min\_support$, it is unnecessary to extend s any more.

3.2. Multi-Pass Mining and Support Threshold Raising

Since our task is to mine top-k closed sequential patterns without $min\_support$ threshold, the mining process should start with $min\_support = 1$, raise it progressively during the mining process, and then use the raised $min\_support$ to prune the search space. This can be done as follows: *as soon as at least k closed sequential patterns with length no less than min_ℓ are found, $min\_support$ can be set to the support of the least frequent pattern, and this $min\_support$-raising process continues throughout the mining process.*

This $min\_support$-raising technique is simple and can lead to efficient mining. However, there are two major problems that need to be addressed. The first is how to verify whether a newly found pattern is closed. This will be discussed in subsection 3.3. The second is how to raise $min\_support$ as quickly as possible. When $min\_support$ is initiated or is very low, the search space will be huge and it is likely to find many patterns with pretty low support. This will lead to the slow raise of $min\_support$. As a result, many patterns with low support will be mined first but be discarded later when enough patterns with higher support are found. Moreover, since a user is only interested in patterns with length at least min_ℓ, many of the projected databases built at levels above min_ℓ may not produce any frequent patterns at level min_ℓ and below. Therefore, a naïve mining algorithm that traverses the search space in lexicographic order will make the mining of the top-k closed sequential patterns very slow. Breadth-first search also does not work. Since short sequences may not have high frequency, one has to access lots of useless low support short sequences before finding any high support long sequences.

In this section we propose a heuristic search space traversal algorithm which in most cases mines the top-k frequent patterns as quickly as the currently fastest sequential patterns mining algorithm, even when the latter is tuned with the most appropriate $min\_support$ threshold.

3.2.1. Optimal traversal of the search space

First, let us define what we mean by optimal traversal of the search space for top-k mining. Assuming that we have found the k most frequent closed sequential patterns for a given database, we call the support of the least frequent pattern $final\_support$. This is the maximum $min\_support$ that one can raise during the mining process. In Example 1, $final\_support = 3$.

For the purpose of top-k mining, the *optimal traversal of the search space* (i.e., the prefix search tree) is such a traversal that does not visit any node (or projected database) that has support less than $final\_support$, i.e., if the $final\_support$ is given, PrefixSpan will traverse the search space optimally in terms of our top-k mining problem. Figure 3 shows a prefix search tree constructed during an optimal traversal of the search space for Example 1. It is important to note that optimality is defined here only for the basic search space traversal algorithm that will be used in our top-k mining method. The search

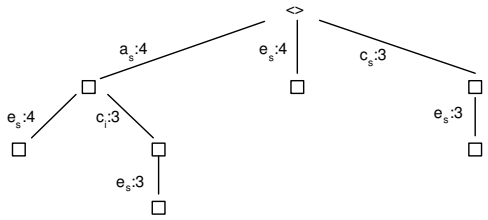


Fig. 3. Optimal Prefix Search Tree

space can be pruned further using other techniques such as the min_ℓ constraint and the early termination conditions discussed in the next subsections. This subsection develops the base algorithm that can traverse the search space as efficiently as if it was given the final support, and then we will build up additional optimization techniques on it.

Algorithm 3.2 GreedyTraversal

Input: A sequence s, $min\_support$, min_ℓ, k, and a projected DB D_s.
Output: The top-k frequent sequence set F.

1: **if** $support(s) < min\_support$ **then return**;
2: **if** $l(s) = min\_\ell$ **then**
3: **Call** PrefixSpanWithSupportRaising(s, $min\_support$, k, D_s, F);
4: **return**;
5: **while** there exists an α in D_s such that $s \diamond_i \alpha$ or $s \diamond_s \alpha$ is the most frequent sequential pattern in the whole database **do**
6: GreedyTraversal($s \diamond_i \alpha$, $min\_support$, min_ℓ, k, $D_{s \diamond_i \alpha}$, F); or
GreadyTraversal($s \diamond_s \alpha$, $min\_support$, min_ℓ, k, $D_{s \diamond_s \alpha}$, F);
7: **return**;

Algorithm 3.2 is a hypothetical algorithm that traverses the first min_ℓ levels of search space greedily without raising $min\_support$. The patterns of length less than min_ℓ will not contribute to the result set, thus they cannot be used to raise $min\_support$. Algorithm 3.2 runs in the way that it always picks the most promising branch in the prefix search tree and does depth-first search. A promising branch means there are lots of closed sequences of length longer than min_ℓ in this branch and their support is very high. We can set different criteria to measure which branch may be promising. Here, we select the pattern which is the most frequent one among all patterns having the same length. After the algorithm reaches the level min_ℓ node, the algorithm calls PrefixSpan to mine the descendant nodes completely. At the same time, it raises $min\_support$ using the method described above.

Algorithm 3.2 is impractical. Notice that the first call to GreedyTraversal has to set $min\_support = 1$ at the beginning. That is, one has to mine patterns with $min\_support = 1$ and build project databases for them before finding the first k closed sequences of length min_ℓ. This is inefficient. Moreover, it is difficult

to implement the criteria shown in line 5. In the next subsection, we propose a multi-pass, heuristic-based mining algorithm, which mitigates these problems.

3.2.2. Multi-pass mining and projected-database tree

Our goal is to develop an algorithm that builds as few prefix-projected databases with support less than $final\_support$ as possible. Actually, we can first search the most promising branches in the prefix search tree in Figure 2 and use the raised $min\_support$ to search the remaining branches. The algorithm is outlined as follows: (1) initially (during the first pass), build a small, limited number of projected databases for each prefix length, $l(l < min\_\ell)$, (2) then (in the succeeding passes) gradually relax the limitation on the number of projected databases that are built, and (3) repeat the mining again. Each time when we reach a projected database D_s, where $l(s) = min\_\ell - 1$, we mine D_s completely and use the mined sequences to raise $min\_support$. The stop condition for this multi-pass mining process is when all projected databases at level min_ℓ with support greater than $min\_support$ are mined completely. We limit the number of projected databases constructed at each level by setting different support thresholds for different levels. The reasoning behind this is that if we set a support threshold that is passed by a small number of projected databases at some higher level, in many cases this support will not be passed by any projected databases at lower levels and vice versa.

Algorithm 3.3 TopSequencesTraversal

Input: A sequence s, $min\_support$, min_ℓ, k, a projected DB D_s,
histograms H[1..min_ℓ], and constant factor θ.
Output: The top-k frequent sequence set F.
1: **if** $support(s) < min\_support$ **then return**;
2: **if** $l(s) = min\_\ell$ **then**
3: **Call** PrefixSpanWithSupportRaising(s, $min\_support$, k, D_s, F);
4: **return**;
5: scan D_s once, find every frequent item α such that
(a) s can be extended to $(s \diamond_i \alpha)$, or
(b) s can be extended to $(s \diamond_s \alpha)$;
insert item α into histogram H[$l(s) + 1$];
6: $next\_level\_top\_support \leftarrow$ GetLevelTopSupportFromHistogam(θ, H[$l(s) + 1$])
7: **for each** α, $support(\alpha) \geq next\_level\_top\_support$ **do**
8: TopSequencesTraversal($s \diamond_i \alpha$, $min\_support$, min_ℓ, k, $D_{s\diamond_i\alpha}$, H, θ); or
TopSequencesTraversal($s \diamond_s \alpha$, $min\_support$, min_ℓ, k, $D_{s\diamond_s\alpha}$, H, θ);
9: **return**;

Algorithm 3.3 performs a single pass of TSP. In order to find the complete result set we need to call this algorithm multiple times to cover all potential branches. The limit on the number of projected databases that are built during each pass is enforced by function GetLevelTopSupportFromHistogam, which uses histograms of the supports of the sequences found earlier in the same pass or in the previous passes and the factor θ which is set in the beginning of each pass. Figure 4 illustrates the multi-pass mining on the problem setting from Example 1, the bolded lines show the branches traversed in each pass. In this example,

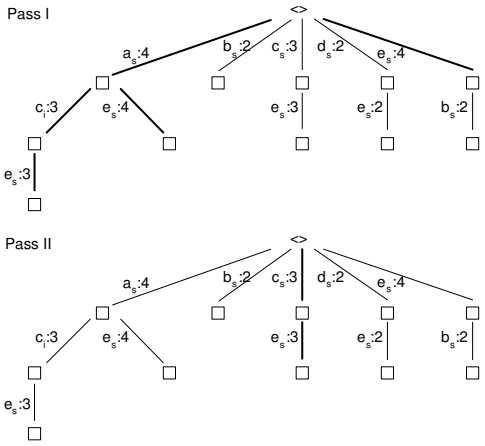


Fig. 4. Multi-pass Mining

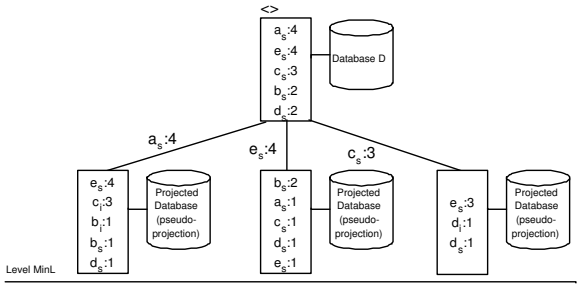


Fig. 5. PDB-tree: A Tree of Prefix-projected Databases

the mining is completed after the second pass because after this pass the support threshold is raised to 3 and there are no unvisited branches with support greater than or equal to 3.

In our current implementation the factor θ is a percentile in the histograms and the function GetLevelTopSupportFromHistogam returns the value of the support at θ-th percentile in the histogram. The initial value of θ is calculated in the beginning of the mining process using the following formula: $\theta = (k * minl)/N_{Items}$, where N_{Items} is the number of distinct items in the database. In each of the following passes the value of θ is doubled. Our experiments show that the performance of the top-k mining algorithm does not change significantly for different initial values of θ as long as they are small enough to divide the mining process in several passes.

In order to efficiently implement the multi-pass mining process described above we use a tree structure that stores the projected databases built in the previous passes. We call this structure *Projected Database Tree* or *PDB-tree*. The PDB-tree is a memory representation of the prefix search tree and stores information about partially mined projected databases during the multi-pass mining process. Since the PDB-tree consists of partially mined projected databases, once

a projected database is completely mined, it can be removed from the PDB-tree. Because of this property, the PDB-tree has a significantly smaller size than the whole prefix search tree traversed during the mining process. The maximum depth of the PDB-tree is always less than min_ℓ because TSP mines all projected databases at level min_ℓ and below completely. In order to further reduce the memory required to store the PDB-tree, we use pseudo-projected databases at the nodes of the PDB-tree, i.e., we only store lists of pointers to the actual sequences in the original sequence database. Figure 5 shows an example of PDB-tree, where each searched node is associated with a projected database.

3.3. Verification of Closed Patterns

Now we come back to the question raised earlier in this section: how can we guarantee that at least k *closed patterns* are found so that $min\_support$ can be raised in mining? Currently, CloSpan mines closed sequential patterns. CloSpan stores candidates for closed patterns during the mining process and in its last step it finds and removes the non-closed ones. This approach is infeasible in top-k mining since it needs to know which pattern is closed and accumulates at least k closed patterns before it starts to raise the minimum support. Thus closed pattern verification cannot be delayed to the final stage.

In order to raise $min\_support$ correctly, we need to maintain a result set to ensure that there exists no pattern in the database that can absorb more than one pattern in the current result set. Otherwise, if such a pattern exists, it may reduce the number of patterns in the result set down to below k and make the final result incomplete or incorrect. For example, assume $k = 2, min\_\ell = 2$, and the patterns found so far are: $\{\langle(a),(b)\rangle : 5, \langle(a),(c)\rangle : 5\}$. If these patterns are used to raise $min\_support$ to 5 but later a pattern $\langle(a),(b),(c)\rangle : 5$ is found, the latter will absorb the first two. Then the result set will consist of only one pattern instead of two. Thus it is incorrect to set $min\_support$ to 5. In this case the correctness and completeness of the final result can be jeopardized because during some part of the mining one might have used an invalid support threshold.

Here we present a technique that handles this problem efficiently.

Definition 3.4. Given a sequence s, $s \in D$, the set of the sequence IDs of all sequences in the database D that contain s is called **sequence ID list**, denoted by $SIDList(s)$. The sum of $SIDList(s)$ is called **sequence ID sum**, denoted by $SIDSum(s)$.

If the sequences in the original database D do not have numeric identification numbers, we can assign such numbers when we scan the database.

Remark 3.1. Given sequences s' and s'', if $s' \sqsubseteq s''$ and $support(s') = support(s'')$ then $SIDList(s') = SIDList(s'')$ and $SIDSum(s') = SIDSum(s'')$.

Rationale. Since s' is a subsequence of s'', s' is contained in all sequences in the database that contain s''. Also, s' cannot be contained in any sequences that do not contain s'' because s' and s'' have the same support. Therefore, $SIDList(s') = SIDList(s'')$ and $SIDSum(s') = SIDSum(s'')$. ∎

Lemma 3.1. Given sequences s' and s'', if $support(s') = support(s'')$, $SIDList(s') \neq SIDList(s'')$, then neither s' is subpattern of s'', nor s'' is a subpattern of s'.

Rationale. This lemma can be easily proved by contradiction using Remark 3.1. Assume $s' \sqsubseteq s''$. Given that $support(s') = support(s'')$, we have $SIDList(s') = SIDList(s'')$ from Remark 3.1, which is a contradiction. In the same way we can prove that $s'' \sqsubseteq s'$ is not possible either. ■

Remark 3.2. If there exists a frequent item $\alpha, \alpha \in D_s$, such that $support(s \diamond_s \alpha) = support(s)$ or $support(s \diamond_i \alpha) = support(s)$, then s should not be added to the current top-k result set, because there exists a superpattern of s with the same support.

Rationale. Since $support(s \diamond_s \alpha) = support(s)$ and $s \sqsubset (s \diamond_s \alpha)$, thus s is not a closed pattern and should not be added to the current result set. Similarly, we can prove it for the case of itemset extension. ■

Based on Remarks 3.1 and 3.2 and Lemma 3.1, we developed an efficient verification mechanism to determine whether a pattern should be added to the top-k set and whether it should be used to raise the support threshold.

A prefix tree, called $TopK\_Tree$, is developed to store the current top-k result set in memory. Also, in order to improve the efficiency of the closed pattern verification, a hash table, called $SIDSum\_Hash$, is maintained that maps sequence id sums to the nodes in $TopK\_Tree$.

In our top-k mining algorithm when a new pattern is found the algorithm takes one of the following three actions: (1) *add_and_raise*: the pattern is added to the top-k result set and is used to raise the support threshold, (2) *add_but_no_raise*: the pattern is added to the top-k result set but is not used to raise the support threshold, and (3) *no_add*: the pattern is not added to the top-k result set.

The following algorithm implements the closed pattern verification.

Algorithm 3.4 Closed Pattern Verification

Input: A sequential pattern s
Output: One of the following three operations: add_and_raise, add_but_no_raise, and no_add.

```
1: if ∃ an item α, such that support(s ⋄_s α) = support(s) or
      support(s ⋄_i α) = support(s) then
      return(no_add);
2: if SIDSum(s) is not in SIDSum_Hash then
      return(add_and_raise);
3: for each s' such that SIDSum(s') = SIDSum(s) and
   Support(s') = Support(s) do
4:    if s ⊏ s' then return(no_add);
5:    if s' ⊏ s then
6:          replace s' with s;
7:          return(add_but_no_raise);
8:    if SIDList(s') = SIDList(s) then
9:          return(add_but_no_raise);
10: return(add_and_raise);
```

Notice that the algorithm for closed pattern verification returns add_but_no_raise for patterns that have the same SIDList as some other patterns that are

already in the top-k result set. Such patterns are stored separately and are not used to raise the support threshold $min\_support$. This eliminates the problem mentioned earlier: If two patterns in the top-k result set are absorbed by a single new pattern, it may lead to less than k patterns in the result set. In summary, our strategy is to maintain top-k patterns in the result set where no two patterns can be absorbed by a single new pattern.

3.4. Applying the Minimum Length Constraint

Now we discuss how to reduce the search space using the minimum length constraint min_ℓ.

Remark 3.3. (Minimum Length Constraint) For any sequence $s' \in D_s$ such that $l(s') + l(s) < min\_\ell$, the sequence s' will not contribute to a frequent sequential pattern of minimum length min_ℓ, and it can be removed from the projected database D_s.

Based on Remark 3.3, when our algorithm builds a projected database, it checks each projected sequence to see whether it is shorter than $min\_\ell - l(s)$ before adding it to the projected database.

Notice that the minimum length constraint can be used to reduce the size of a projected database D_s only when $l(s) < min\_\ell - 1$. Thus when the prefix s is longer than $min\_\ell - 2$, the program does not need to check the length of the projected sequences.

3.5. Early Termination by Equivalence

Early termination by equivalence is a search space reduction technique developed in CloSpan(Yan et al, 2003). Let $\mathcal{I}(D)$ represent the total number of items in D, defined as

$$\mathcal{I}(D) = \sum_{i=1}^{n} l(s_i).$$

We call $\mathcal{I}(D)$ the *size of the database*. For the sample dataset in Table 1, $\mathcal{I}(D) = 17$. The property of early termination by equivalence shows if two sequences $s \sqsubseteq s'$ and $\mathcal{I}(D_s) = \mathcal{I}(D_{s'})$, then $\forall \gamma, support(s \diamond \gamma) = support(s' \diamond \gamma)$. It means the descendants of s in the lexicographical sequence tree must not be closed. Furthermore, the descendants of s and s' are exactly the same. CloSpan uses this property to quickly prune the search space of s.

To facilitate early termination by equivalence in the top-k mining, we explore both the partially mined projected database tree, PDB_Tree, and the result set tree, $TopK\_Tree$. Two hash tables are maintained: one, called PDB_Hash, mapping databases sizes to nodes in PDB_Tree and the other, called $TopK\_Hash$, mapping databases sizes to nodes in $TopK\_Tree$.

For each new projected database D_s that is built, we search the two hash tables using $\mathcal{I}(D_s)$ as a key and check the following conditions:

- If there exists a sequence $s', s' \in PDB\_Tree$, such that $\mathcal{I}(D_s) = \mathcal{I}(D_{s'})$ and $s \sqsubseteq s'$ then stop the search of the branch of s.

abbr.	meaning
D	Number of sequences in 000s
C	Average itemsets per sequence
T	Average items per itemset
N	Number of different items in 000s
S	Average itermsets in maximal sequences
I	Average items in maximal sequences

Table 2. Parameters in IBM Quest Data Generator

- If there exists a sequence $s', s' \in PDB\_Tree$, such that $\mathcal{I}(D_s) = \mathcal{I}(D_{s'})$ and $s' \sqsubseteq s$ then remove s' from PDB_Tree and continue the mining of the branch of s.
- If there exists a sequence $s', s' \in TopK\_Tree$ and $s' \notin PDB\_Tree$, such that $\mathcal{I}(D_s) = \mathcal{I}(D_{s'})$ and $s \sqsubseteq s'$ then stop the search of the branch of s.

With this adoption of early termination in TSP, the performance of TSP is improved significantly.

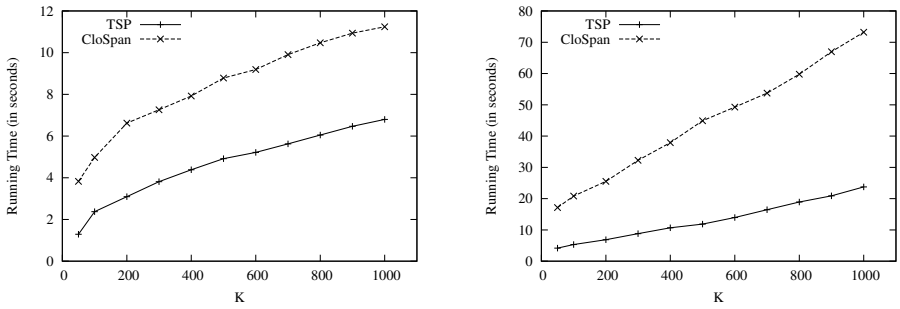
4. Experimental Evaluation

This section reports the performance testing of TSP in large data sets. In particular, we compare the performance of TSP with CloSpan. The comparison is based on assigning the optimal *min_support* to CloSpan so that it generates the same set of top-k closed patterns as TSP for specified values of k and min_ℓ. The optimal *min_support* is found by first running TSP under each experimental condition. Since this optimal *min_support* is hard to speculate without mining, even if TSP achieves the similar performance with CloSpan, TSP is still more valuable since it is much easier for a user to work out a k value for top-k patterns than a specific *min_support* value.

The datasets used in this study are generated by a synthetic data generator provided by IBM. It can be obtained at http://www.almaden.ibm.com/cs/quest. Table 2 shows the major parameters that can be specified in this data generator, more details are available in (Agrawal and Srikant, 1995).

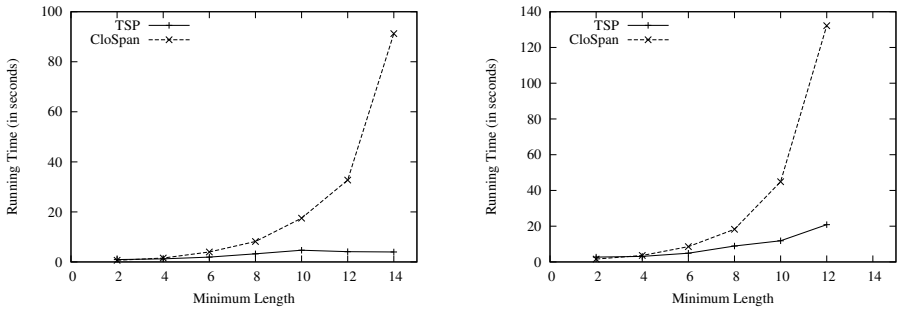
All experiments were performed on a 1.8GHz Intel Pentium-4 PC with 512MB main memory, running Windows XP Professional. Both algorithms are written in C++ using STL and compiled with Visual Studio .Net 2002.

The performance of the two algorithms has been compared by varying min_ℓ and k. When k is fixed, its value is set to either 50 or 500 which covers the range of typical values for this parameter. Figures 6 and 7 show performance results for dataset D100C5T2.5N10S4I2.5. This dataset consists of relatively short sequences, each sequence contains 5 itemsets on average and the itemsets have 2.5 items on average. The experimental results show that TSP mines this dataset very efficiently and in most cases runs several times faster than CloSpan. The difference between the running time of the two algorithms is more significant when longer patterns are mined (larger min_ℓ). There are two major reasons



(a) $min\_\ell = 6$ (b) $min\_\ell = 10$

Fig. 6. Dataset D100C5T2.5N10S4I2.5, min_ℓ



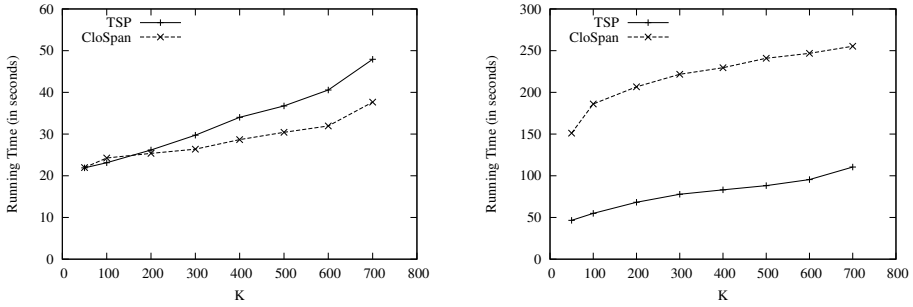
(a) $K = 50$ (b) $K = 500$

Fig. 7. Dataset D100C5T2.5N10S4I2.5, K

for the better performance of TSP in this dataset. First, it uses the min_ℓ constraint to prune short sequences during the mining process which in some cases significantly reduces the search space and improves the performance. Second, TSP has more efficient closed pattern verification scheme and stores a result set that contains only a small number of closed patterns, while CloSpan keeps a larger number of candidate patterns that could not be closed and removes the non-closed ones at the end of the mining processes.

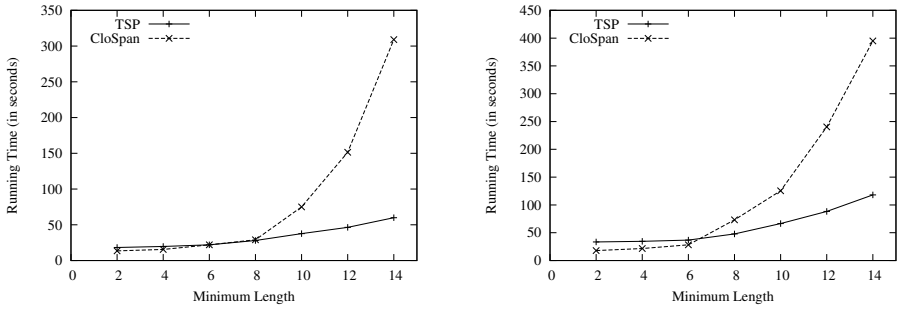
Figures 8 and 9 show the experiments on dataset D100C10T10N10S4I5 which consists of longer patterns compared to the previous one. The average number of itemsets per sequence in this dataset is increased from 5 to 10. For this dataset the two algorithms have comparable performance when min_ℓ is small. The reasons for the similar performance of the two algorithms are that the benefit of applying the min_ℓ constraint is smaller because the sequences in the dataset are relatively longer. However, as indicated by Figure 9, when min_ℓ increases, TSP runs faster than CloSpan again.

As we can see, min_ℓ plays an important role in improving the performance of TSP. If we ignore the performance gain caused by min_ℓ, TSP can achieve the competitive performance with well tuned CloSpan. We may wonder why minimum support-raising cannot boost the performance like what min_ℓ does. The rule of thumb is that the support of upper level nodes should be greater than lower level nodes (the support of short sequences should be greater than



(a) $min\_\ell = 6$ (b) $min\_\ell = 12$

Fig. 8. Dataset D100C10T10N10S6I5, min_ℓ



(a) $K = 50$ (b) $K = 500$

Fig. 9. Dataset D100C10T10N10S6I5, K

that of long sequences). Then, few nodes in the upper level can be pruned by the minimum support. Since we cannot access the long patterns without accessing the short patterns, we have to search most of upper level nodes in the prefix search tree. As we know, the projected database of the upper level nodes is very big and expensive to compute. Thus, if we cannot reduce checking the projected databases of the upper level nodes, it is unlikely that we can benefit from support-raising technique a lot. However, the support-raising technique can free us from setting minimum support without sacrificing the performance.

In summary, the multi-pass search space traversal strategy combined with the dynamic raising of $min\_support$ avoid the construction of a large number of unnecessary projected databases with support less than $final\_support$. Even though the top-k algorithm is not given any minimum support threshold, it achieves a similar or even better performance in comparison with CloSpan running with the optimal $min\_support$ threshold.

5. Discussion

In this section, we discuss the related work and the directions for further study.

5.1. Related work

We first show the relationships and differences of the TSP algorithm with (1) the Apriori-based algorithms: AprioriAll, GSP and SPADE, (2) the pattern growth-based algorithms: FreeSpan, PrefixSpan, and CloSpan, and (3) the top-k frequent pattern mining algorithm: TFP.

5.1.1. Apriori-Based Algorithms

Agrawal and Srikant (Agrawal and Srikant, 1995) introduced the sequential pattern mining problem and three algorithms to solve it. Among those algorithms, AprioriAll was the only one to mine the complete set of frequent sequential patterns. Later in (Srikant and Agrawal, 1996) they proposed the GSP (Generalized Sequential Patterns) algorithm for mining sequential patterns. All of these algorithms are based on the Apriori property proposed in association mining (Agrawal and Srikant, 1994) and a candidate generation-and-test approach. The Apriori property states that any superpattern of a nonfrequent pattern cannot be frequent. Using this heuristic, AprioriAll and GSP narrow down the search space for frequent sequential patterns drastically. To mine frequent sequences with length $(l+1)$, the Apriori-based algorithm needs to find all the candidate length-$(l+1)$ sequences from their previously derived length-l frequent sequences, scan the database one more time to collect their counts, which makes them inefficient for mining long patterns.

5.1.2. SPADE

Zaki in (Zaki, 2001) proposed a new approach for mining frequent sequential patterns, called SPADE (Sequential Pattern Discovery using Equivalence classes). This algorithm uses vertical id-list database format, i.e., for each item an id-list of the identifiers of the sequences in which it appears and their corresponding time stamps are created. The frequent sequential patterns are mined by performing temporal join operations on these id-lists. SPADE decomposes the original problem into smaller subproblems, which can be independently solved in main-memory, using lattice search techniques. SPADE outperforms GSP by a factor of two, and by an order of magnitude with some pre-processed data. This method can mine the complete set of frequent sequences in only three database scans.

5.1.3. FreeSpan and PrefixSpan

FreeSpan (Frequent pattern-projected sequential pattern mining) was introduced by Han, et al. in (Han et al, 2000). It uses frequent items to recursively project sequence databases into a set of smaller projected databases. The subsequent mining is limited to each of these smaller projected databases. *FreeSpan* is significantly more efficient than the Apriori-based GSP . The problem of *FreeSpan* is that the same sequence can be repeated in many projected databases. For example, if a sequential pattern appears in each sequence in the database, its projected database will have the same size as the original database, except for the infrequent items that will be removed.

In a later work (Pei et al, 2001) Pei, et al. introduced PrefixSpan (Prefix-projected Sequential Pattern mining). Its general idea is to examine only the prefix subsequences and project only their corresponding suffix subsequences

into projected databases. In each projected database, sequential patterns are grown by exploring only local frequent patterns. PrefixSpan runs considerably faster than both GSP and *FreeSpan* , especially when longer sequential patterns are mined.

5.1.4. CloSpan

CloSpan (Yan et al, 2003) (Closed Sequential Patterns mining) is a recently proposed closed sequence mining algorithm. It uses depth-first search and prefix-projected database method to enumerate the frequent sequential patterns. CloSpan developed a novel technique called early termination by equivalence, which can efficiently determine whether there are new closed patterns in search subspaces and terminate the search of subspaces that do not contain such patterns. CloSpan outperforms PrefixSpan by more than one order of magnitude and is capable of mining longer frequent sequences in large databases with low minimum support. An alternative to closed sequence mining is maximal sequence mining (Chen et al, 1996). A maximal sequence is a frequent sequence that is not contained in any other frequent sequence. One may lose support information on frequent sequences when mining maximal sequences.

5.1.5. TFP*: Mining top-k Frequent Closed Patterns without Minimum Support*

The algorithms reviewed in the last four subsections mine frequent sequential patterns in sequence databases using user-specified *min_support* threshold. Our algorithm has a frequent pattern mining counterpart, TFP (Han et al, 2002), that mines frequent closed itemsets in transaction databases. Even though TFP does not mine sequential patterns, it is closely related to TSP because TFP is the first study on mining top-k frequent closed patterns with minimum length constraint.

TFP is an FP-tree (Han et al, 2000) based frequent pattern mining algorithm for finding the top-k frequent closed patterns without a predefined *min_support* threshold. TFP starts the mining process with *min_support* threshold equal to 1, and raises the support threshold during both the FP-tree construction and the mining of the FP-tree. TFP explores "top-down" and "bottom-up" combined FP-tree mining process to first mine the most promising tree branches. Also, an efficient closure verification scheme is developed to determine whether the newly discovered patterns are closed. TFP in most cases achieves better performance than two of the most efficient frequent closed pattern mining algorithms, CLOSET (Pei et al, 2000) and CHARM (Zaki and Hsiao, 2002), even when they are running with the best tuned *min_support* threshold. TFP concludes that mining the top-k frequent patterns without *min_support* can be efficient and should be more preferable than the traditional *min_support*-based mining.

The algorithm proposed in the present paper adopts the problem definition of TFP and provides an efficient solution to this problem in the more challenging setting of mining frequent closed sequential patterns in sequence databases.

5.2. Future work

There are several issues related to our algorithm for mining top-k frequent closed sequential patterns that should be studied further. For example, more sophisticated methods for determining the parameter θ that controls the level support thresholds in the different passes of the multi-pass mining process should be investigated. It is possible to use the support distributions observed in the earlier passes to set more appropriate values of θ for the subsequent passes. Another potential optimization is to modify the algorithm to perform the breath-first search of the search space after it reaches level min_ℓ. This can avoid traversal of long branches in the beginning of the mining and will raise $min\_support$ faster. However, the breath-first traversal also has its disadvantages: it requires more memory and limits the usage of early termination by equivalence. Thus, a comprehensive study on a variety of datasets needs to be done to evaluate in what situations that the algorithm should use breath-first search instead of depth-first search.

The performance study presented in this study includes only synthetic data sets. In order to better evaluate the scalability and flexibility of the proposed algorithm, experiments on real datasets need to be done. Other directions for future research include incorporation of user-specified constraints (Garofalakis et al, 1999) (Pei et al, 2002) into the mining of top-k closed sequential patterns and extension of the method to mining other complicated structured patterns, such as closed graph patterns (Yan and Han, 2003).

6. Conclusions

In this paper, we have studied the problem of mining top-k (frequent) closed sequential patterns with length no less than min_ℓ and proposed an efficient mining algorithm TSP, with the following distinct features: (1) it adopts a novel, multi-pass search space traversal strategy that allows mining of the most frequent patterns early in the mining process and fast raising of the minimal support threshold $min\_support$ dynamically, which is then used to prune the search space, (2) it performs efficient closed pattern verification during the mining process that ensures accurate raising of $min\_support$ and derives correct and complete results, and (3) it develops several additional optimization techniques, including applying the minimum length constraint, min_ℓ, and incorporating the early termination testing proposed in CloSpan.

Our experimental study shows that the proposed algorithm delivers competitive performance and in many cases outperforms CloSpan, currently the most efficient algorithm for (closed) sequential pattern mining, even when CloSpan is running with the best tuned $min\_support$. Through this study, we conclude that mining top-k closed sequential patterns without $min\_support$ is practical and in many cases more preferable than the traditional $min\_support$ threshold based sequential pattern mining.

Acknowledgements. This work was supported in part by the U.S. National Science Foundation NSF IIS-02-09199, University of Illinois, and a gift from Microsoft Research. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies. The authors are grateful to anonymous reviewers for their very constructive and helpful comments and suggestions on the initial version of the paper.

References

Agrawal R, Srikant R (1994) Fast Algorithms for Mining Association Rules. In Bocca J B, Jarke M, Zaniolo C (Eds.). Proceedings of the 20th International Conference on Very Large Data Bases, Santiago de Chile, Chile, September 1994, pp 487–499

Agrawal R, Srikant R (1995) Mining Sequential Patterns. In Yu P S, Chen A (Eds.). Proceedings of the 11th International Conference on Data Engineering, Taipei, Taiwan, March 1995, pp 3–14

Ayres J, Gehrke J E, Yiu T, Flannick J (2002) Sequential Pattern Mining Using Bitmaps. Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Edmonton, Canada, July 2002, pp 429-435

Chen M S, Park J S, Yu P S (1996) Data Mining for Path Traversal Patterns in a Web Environment. Proceedings of the 16th International Conference on Distributed Computing Systems, Hong Kong, China, May 1996, pp 385–392

Garofalakis M, Rastogi R, Shim K (1999) SPIRIT: Sequential Pattern Mining with Regular Expression Constraints. In Atkinson M P, Orlowska M E, Valduriez P, Zdonik S B, Brodie M L (Eds.). Proceedings of the 25th International Conference on Very Large Data Bases, Edinburgh, Scotland, September 1999, pp 223–234

Guha S, Rastogi R, Shim K (1999) ROCK: A Robust Clustering Algorithm for Categorical Attributes. Proceedings of the 15th International Conference on Data Engineering, Sydney, Austrialia, March 1999, pp 512–521

Han J, Pei J, Mortazavi-Asl B, Chen Q, Dayal U, Hsu M C (2000) FreeSpan: Frequent Pattern-Projected Sequential Pattern Mining. Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Boston, MA, August 2000, pp 355–359

Han J, Pei J, Yin Y (2000) Mining Frequent Patterns without Candidate Generation. In Chen W, Naughton J F, Bernstein P A (Eds.). Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, TX, May 2000, pp 1-12

Han J, Wang J, Lu Y, Tzvetkov P (2002) Mining Top-K Frequent Closed Patterns without Minimum Support. Proceedings of the 2002 IEEE International Conference on Data Mining Maebashi City, Japan, December 2002, pp 211–218

Li Z, Srinivasan S M, Chen Z, Zhou Y, Tzvetkov P, Yan X, Han J (2003) Using Data Mining for Discovering Patterns in Autonomic Storage Systems. Proceedings of the 2003 ACM Workshop on Algorithms and Architectures for Self-Managing Systems, San Diego, CA, June 2003

Mannila H, Toivonen H, Verkamo A I (1995) Discovering Frequent Episodes in Sequences. In Fayyad U, Uthurusamy R (Eds.). Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD-95), Montreal, Canada, August 1995, pp 210–215

Srikant R, Agrawal R (1996) Mining Sequential Patterns: Generalizations and Performance Improvements. In Apers P, Bouzeghoub M, Gardarin G (Eds.). Advances in Database Technology - EDBT'96, 5th International Conference on Extending Database Technology, Avignon, France, March 1996, pp 3–17

Pei J, Han J, Mao R (2000) CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets. In Gunopulos D, Rastogi R (Eds.). 2000 ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, Dallas, TX, May 2000, pp 11–20

Pei J, Han J, Mortazavi-Asl B, Pinto H, Chen Q, Dayal U, Hsu M C (2001) PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth. Proceedings of the 17th International Conference on Data Engineering, Heidelberg, Germany, April 2001, pp 215–224

Pei J, Han J, Wang W (2002) Constraint-Based Sequential Pattern Mining in Large Databases. Proceedings of the 2002 ACM CIKM International Conference on Information and Knowledge Management, McLean, VA, November 2002, pp 18–25

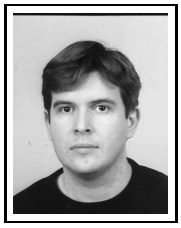
Yan X, Han J, Afshar R (2003) CloSpan: Mining Closed Sequential Patterns in Large Datasets. In Barbar D, Kamath C (Eds.). Proceedings of the 3rd SIAM International Conference on Data Mining, San Francisco, CA, May 2003, pp 166–177

Yan X, Han J (2003) CloseGraph: Mining Closed Frequent Graph Patterns. In Getoor L, Senator T E, Domingos P, Faloutsos C (Eds.). Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, August 2003, pp 286–295

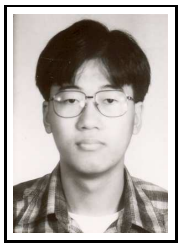
Zaki M (2001) SPADE: An Efficient Algorithm for Mining Frequent Sequences. Machine Learning 40:31–60

Zaki M and Hsiao C (2002) CHARM: An Efficient Algorithm for Closed Itemset Mining. In Grossman R L, Han J, Kumar V, Mannila H, Motwani R (Eds.). Proceedings of the 2nd SIAM International Conference on Data Mining, Arlington, VA, April 2002, pp 457–473

Author Biographies



Petre Tzvetkov is currently an IT manager and software architect at MOST Computers Ltd. His main professional and research interests are in data mining, databases, and software engineering. He has completed his Master of Science degree in Computer Science at University of Illinois under the supervision of Professor Jiawei Han. Petre Tzvetkov has co-authored several research papers in the area of data mining.



Xifeng Yan received a B.E. degree in computer engineering from Zhejiang University, China, in 1997, and an M.S. degree in computer science from the State University of New York at Stony Brook, NY, in 2001. He is currently a Ph.D. student in the Department of Computer Science at the University of Illinois at Urbana-Champaign. His research interests include data mining, structural/graph pattern mining, and their applications in database systems and bioinformatics.

Jiawei Han is a professor in the Department of Computer Science, University of Illinois at Urbana-Champaign. He has been working on research into data mining, data warehousing, database systems, with over 250 conference and journal publications. He has chaired or served on the PCs in many international conferences, including ACM SIGKDD, ACM SIGMOD, VLDB, ICDE, ICDM, SDM, and EDBT. He also served or is serving on the editorial boards for Data Mining and Knowledge Discovery, IEEE Transactions on Knowledge and Data Engineering, and Journal of Intelligent Information Systems. He is currently serving on the Board of Directors for the Executive Committee of ACM Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD). Jiawei has received the Outstanding Contribution Award at the 2002 ICDM, ACM Service Award, and IBM Faculty Awards. He is an ACM Fellow and the first author of the textbook "Data Mining: Concepts and Techniques" (Morgan Kaufmann, 2001).

Correspondence and offprint requests to: Xifeng Yan, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA. Email: xyan@cs.uiuc.edu