TSP: Mining Top-K Closed Sequential Patterns *

Petre Tzvetkov Xifeng Yan Jiawei Han
Department of Computer Science
University of Illinois at Urbana-Champaign, Illinois, U.S.A.
{tzvetkov, xyan, hanj}@cs.uiuc.edu

Abstract

Sequential pattern mining has been studied extensively in data mining community. Most previous studies require the specification of a minimum support threshold to perform the mining. However, it is difficult for users to provide an appropriate threshold in practice. To overcome this difficulty, we propose an alternative 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 minimum length of each pattern. We mine closed patterns since they are compact representations of frequent patterns.

We developed an efficient algorithm, called TSP*, which 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 *outperforms the closed sequential pattern mining algorithm even when the latter is running with the best tuned minimum support threshold.*

1 Introduction

Sequential pattern mining is an important data mining task that has been studied extensively [1, 5, 3, 7, 11, 2]. It was first introduced by Agrawal and Srikant in [1]: *Given a set of sequences, where each sequence consists of a list of itemsets, and given a user-specified minimum support threshold (min_support), sequential pattern mining is to find all frequent subsequences whose frequency is no less than min_support*. This mining task 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 phenomena also exists in itemset and graph patterns when the patterns are large. For example, assume the database contains a frequent sequence $\langle (a_1)(a_2)\ldots(a_{64})\rangle$ $(\forall i \neq j, a_i \neq a_j)$, it will generate $2^{64}-1$ frequent subsequences. It is very likely some subsequences share the exact same support with this long sequence, which are essentially redundant patterns.

Second, 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 answer found*. To come up with an appropriate $min_support$, one needs to have prior knowledge about the mining query and the task-specific data, and be able to estimate beforehand how many patterns will be generated with a particular threshold.

A solution to the first problem was proposed recently by Yan, et al. [10]. Their algorithm, called CloSpan, can mine *closed sequential patterns*. 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, a similar situation occurs in frequent itemset mining. As proposed in [4], a good solution is to change the task of mining frequent patterns to *mining top-k frequent closed patterns of minimum 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 minimum length of the closed patterns. This setting is also desirable in the context of sequential pattern mining. Unfortunately, most of the techniques developed in [4] 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. Nevertheless, some ideas developed in [4] are still influential in our algorithm design.

*The work was supported in part by National Science Foundation under Grant No. 02-09199, the Univ. of Illinois, and Microsoft Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

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 $min_support$ 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 the desired number of 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 [10].

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, the basic concepts of sequential pattern mining are introduced and the problem of mining the top-k closed 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 conclude this study in Section 6.

2 Problem Definition

This section defines the basic concepts in sequential pattern mining, and then formally introduces the problem of mining the top-k closed sequential patterns. We adopt the notations used in [10].

Let $I = \{i_1, i_2, \ldots, i_k\}$ be a set of 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. We assume there exists a linear order in I and items in each itemset are sorted. The length of s, $l(s)$, is the total number of items in s. 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 by $\alpha \sqsubseteq \beta$, 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.

A sequence database, $D = \{s_1, s_2, \ldots, s_n\}$, is a set of sequences. 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 **frequent 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 minimum 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. ■

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

Example 1 Table 1 shows a sample sequence database. We refer to this databases 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: $\langle(ac)\rangle : 3, \langle(c)(e)\rangle : 3$, they are not in the result set because they are not closed and both of them are absorbed by $\langle(ac)(e)\rangle : 3$. ■

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

Table 1. Sample Sequence Database D

3 Method Development

Our method of mining is developed in this section. First, the concept of projection-based sequential pattern mining, PrefixSpan[7], is introduced, 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, two additional optimization techniques are proposed to further improve the efficiency of the algorithm.

3.1 Projection-based Sequential Pattern Mining

Definition 3.1 *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*

[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.

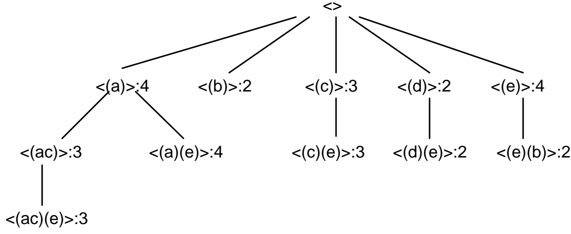


Figure 1. Lexicographic Sequence Tree

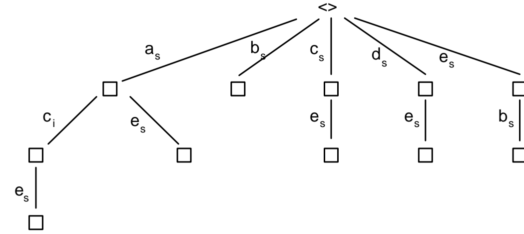


Figure 2. Prefix Search Tree

$\forall u \in t_m, v \in t'_1, u < v$; 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' [10].* ■

For example, $\langle(ae)\rangle$ is an itemset-extension of $\langle(a)\rangle$, whereas $\langle(a)(c)\rangle$ is a sequence-extension of $\langle(a)\rangle$. $\langle(ac)\rangle$ is a prefix of $\langle(ac)(d)(e)\rangle$ and $\langle(d)(e)\rangle$ is its suffix.

Definition 3.2 *An* ***s-projected*** *database is defined as $D_s = \{p \mid s' \in D, s' = r \diamond p$, s.t. 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, $D_{\langle(ac)\rangle} = \{\langle(d)(e)\rangle, \langle(_d)(e)\rangle, \langle(_e)\rangle\}$, where$(_d)$ means that d and the item c in $\langle(ac)\rangle$ come from the same itemset.

Sequence Lexicographic Order is given as follows: (i) if $s' = s \diamond p$, then $s < s'$; (ii) if $s = \alpha \diamond_i p$ and $s' = \alpha \diamond_s p'$, then $s < s'$; (iii) if $s = \alpha \diamond_i p$, $s' = \alpha \diamond_i p'$, and $p < p'$, then $s < s'$; (iv) if $s = \alpha \diamond_s p$, $s' = \alpha \diamond_s p'$, and $p < p'$ then $s < s'$; and (v) if $s = \langle(u)\rangle \diamond p$, $s' = \langle(v)\rangle \diamond p'$, and $u < v$, then $s < s'$ (u and v are the smallest item in the first itemset of s and s' respectively).

For example, $\langle(a, f)\rangle < \langle(b, f)\rangle$, $\langle(ab)\rangle < \langle(ab)(a)\rangle$ (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).

A *Lexicographic Sequence Tree* is constructed as follows: Each node in the tree corresponds to a sequence; each node is either an itemset-extension or sequence-extension of its parent; and the left sibling is less than the right sibling in sequence lexicographic order.

Figure 1 shows a lexicographic sequence tree which records the frequent patterns of the sample database (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 performing an itemset-extension or a sequence-extension.

PrefixSpan [7] 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 itemset-extension and sequence-extension recursively until all the frequent sequences with prefix s are discovered.

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 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 Section 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 spaces in sequence lexicographic order will make the mining of the top-k closed sequential patterns very slow.

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 Multi-pass mining and projected-database tree

Algorithm 3.1 TopSequencesTraversal

Input: A sequence s, a projected DB D_s, min_ℓ,
histograms H[1..min_ℓ], and constant factor θ
Output: The top-k frequent sequence set T.
1: **if** $support(s) < min_support$ **then return**
2: **if** $l(s) = min_\ell$ **then**
3: **Call** PrefixSpanWithSupportRaising(s, D_s, $min_support$, T);
4: **return**;
5: scan D_s once, find every frequent item α such that s can be extended to $s \diamond \alpha$;
insert α in histogram H[$l(s) + 1$];
6: sort items in H[$l(s) + 1$] based on their support;
7: $next_level_top_support \leftarrow$ GetTopSupportFromHistogam (θ, H[$l(s) + 1$])
8: **for each** α, $support(\alpha) \geq next_level_top_support$ **do**
9: **Call** TopSequencesTraversal($s \diamond \alpha$, $D_{s\diamond\alpha}$, min_ℓ);
10: **return**;

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.

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 3.1 performs a single pass of TSP. Line 2-4 calls PrefixSpan to find frequent patterns which have prefix s. Once there are at least k closed patterns discovered, it raises the minimum support threshold to the support of the least frequent one. We call this procedure PrefixSpanWithSupportRaising. In order to find the complete result set we need to call Algorithm 3.1 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 GetTopSupportFromHistogam, 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 3 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, 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.

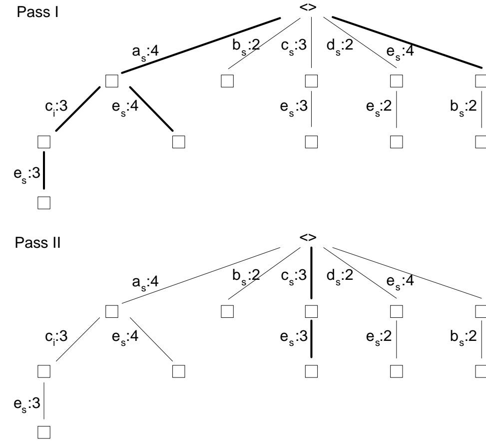


Figure 3. Multi-pass mining

In our current implementation the factor θ is a percentile in the histograms and the function GetTopSupportFromHistogam 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

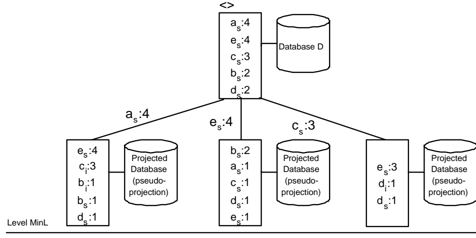


Figure 4. PDB-tree: A tree of prefix-projected databases

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 [7] 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 4 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 there is only one other algorithm, CloSpan, that 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 2. Thus it is not correct 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 3.3 *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 $SIDSum(s)$.*

We have the following results:

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

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'.*

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

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.

Algorithm 3.3 implements closed pattern verification. Notice that Algorithm 3.3 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

Algorithm 3.2 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** $\exists$ an item u, such that $support(s \diamond_i \langle(u)\rangle) = support(s)$ or $support(s \diamond_s \langle(u)\rangle) = support(s)$ **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** $s \sqsubset s'$ **then return**(no_add);
5:   **if** $s' \sqsubset s$ **then**
6:     replace s' with s;
7:     **return**(add_but_no_raise);
8:   **if** $SIDList(s') \equiv SIDList(s)$ **then**
10:     **return**(add_but_no_raise);
11: **return**(add_and_raise);

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 [10]. 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, $\mathcal{I}(D) = 16$. 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.
- 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

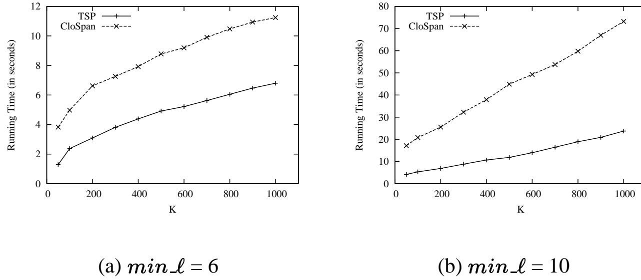
This section reports the performance testing of TSP in large data sets. In particular, we compare the performance of TSP with CloSpan. The comparison is based on assigning the optimal $min_support$ to CloSpan so that it generates the same set of top-k closed patterns as TSP for specified values of k 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 [1].

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 itemsets in maximal patterns
I	Average items in maximal patterns

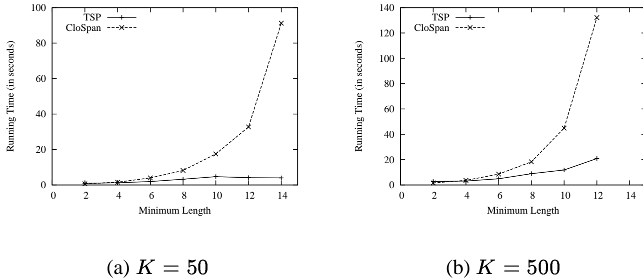
Table 2. Synthetic Data Parameters

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.



(a) $min_\ell = 6$ (b) $min_\ell = 10$

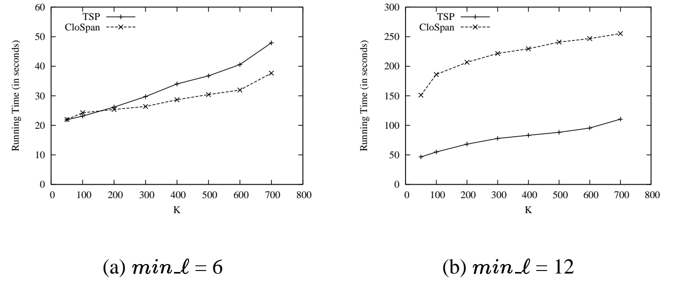
Figure 5. Dataset D100C5T2.5N10S4I2.5



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

Figure 6. Dataset D100C5T2.5N10S4I2.5

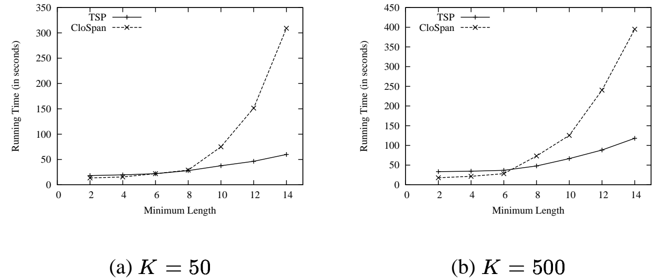
The performance of the two algorithms has been compared by varying min_ℓ and k. When k is fixed, its value is set to either 50 or 500 which covers the range of typical values for this parameter. Figures 5 and 6 show performance results for dataset D100C5T2.5N10S4I2.5. This dataset consists of relatively 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 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.



(a) $min_\ell = 6$ (b) $min_\ell = 12$

Figure 7. Dataset D100C10T10N10S6I5

Figures 7 and 8 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. 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 longer, and also the major cost for mining this dataset is the construction of prefix-projected databases which has similar implementation in both algorithms.



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

Figure 8. Dataset D100C10T10N10S6I5

As we can see, min_ℓ plays an important role in improving the performance of TSP. If we ignore the perfor-

mance 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 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 upper level nodes' projected databases, it is unlikely 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.

5 Related Work

Agrawal and Srikant [1] introduced the sequential pattern mining problem. Efficient algorithms like GSP [8], SPADE [11], PrefixSpan [7], and SPAM [2] were developed. Because the number of frequent patterns is too huge, recently several algorithms were proposed for closed pattern mining: CLOSET [6] and CHARM [12] for closed itemset mining, CloSpan [10] for closed sequence mining, and CloseGraph [9] for closed graph mining. All of these algorithms can deliver much less patterns than frequent pattern mining, but do not lose any information. Top-k closed pattern mining intends to reduce the number of patterns further by only mining the most frequent ones.

As to top-k closed sequential pattern mining, CloSpan [10] and TFP [4] are the most related work. CloSpan mines frequent closed sequential patterns while TFP discovers top-k closed itemsets. 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.

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 minimum 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 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 minimum support threshold based sequential pattern mining.

References

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *ICDE'95*, pp. 3–14, Taipei, Taiwan, Mar. 1995.

[2] J. Ayres, J. E. Gehrke, T. Yiu, and J. Flannick. Sequential pattern mining using bitmaps. In *KDD'02*, pp. 429–435, Edmonton, Canada, July 2002.

[3] S. Guha, R. Rastogi, and K. Shim. Rock: A robust clustering algorithm for categorical attributes. In *ICDE'99*, pp. 512–521, Sydney, Australia, Mar. 1999.

[4] J. Han, J. Wang, Y. Lu, and P. Tzvetkov. Mining top-k frequent closed patterns without minimum support. In *ICDM'02*, pp. 211–218, Maebashi, Japan, Dec. 2002.

[5] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episodes in sequences. In *KDD'95*, pp. 210–215, Montreal, Canada, Aug. 1995.

[6] J. Pei, J. Han, and R. Mao. CLOSET: An effi cient algorithm for mining frequent closed itemsets. In *DMKD'00*, pp. 11–20, Dallas, TX, May 2000.

[7] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Prefi xSpan: Mining sequential patterns effi ciently by prefi x-projected pattern growth. In *ICDE'01*, pp. 215–224, Heidelberg, Germany, April 2001.

[8] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *EDBT'96*, pp. 3–17, Avignon, France, Mar. 1996.

[9] X. Yan and J. Han. CloseGraph: Mining closed frequent graph patterns. In *KDD'03*, Washington, D.C., Aug. 2003.

[10] X. Yan, J. Han, and R. Afshar. CloSpan: Mining closed sequential patterns in large datasets. In *SDM'03*, pp. 166–177, San Fransisco, CA, May 2003.

[11] M. Zaki. SPADE: An effi cient algorithm for mining frequent sequences. *Machine Learning*, 40:31–60, 2001.

[12] M. J. Zaki and C. J. Hsiao. CHARM: An effi cient algorithm for closed itemset mining. In *SDM'02*, pp. 457–473, Arlington, VA, April 2002.