TKS: Efficient Mining of Top-K Sequential Patterns

Philippe Fournier-Viger[1], Antonio Gomariz[2], Ted Gueniche[1], Espérance Mwamikazi[1] and Rincy Thomas[3]

[1]Department of Computer Science, University of Moncton, Canada
[2]Dept. of Information and Communication Engineering, University of Murcia, Spain
[3]Department of Computer Science, SCT, Bhopal, India

{philippe.fournier-viger,etg8697,eem7706}@umoncton.ca,
rinc_thomas@rediffmail.com, agomariz@um.es

Abstract. Sequential pattern mining is a well-studied data mining task with wide applications. However, fine-tuning the *minsup* parameter of sequential pattern mining algorithms to generate enough patterns is difficult and time-consuming. To address this issue, the task of top-*k* sequential pattern mining has been defined, where *k* is the number of sequential patterns to be found, and is set by the user. In this paper, we present an efficient algorithm for this problem named TKS (*Top-K Sequential pattern mining)*. TKS utilizes a vertical bitmap database representation, a novel data structure named PMAP (*Precedence Map*) and several efficient strategies to prune the search space. An extensive experimental study on real datasets shows that TKS outperforms TSP, the current state-of-the-art algorithm for top-*k* sequential pattern mining by more than an order of magnitude in execution time and memory.

Keywords: top-k, sequential pattern, sequence database, pattern mining

1 Introduction

Various methods have been proposed for mining temporal patterns in sequence databases such as mining repetitive patterns, trends and sequential patterns [8]. Among them, sequential pattern mining is probably the most popular set of techniques. Given a user-defined threshold *minsup* and a set of sequences, it consists of discovering all subsequences common to more than *minsup* sequences [1]. It is a well-studied data mining problem with wide applications such as the analysis of web click-streams, program executions, medical data, biological data and e-learning data [5, 8]. Although many studies have been done on designing sequential pattern mining algorithms [1, 2, 3, 8], an important problem is how the user should choose the *minsup* threshold to generate a desired amount of patterns. This problem is important because in practice, users have limited resources (time and storage space) for analyzing the results and thus are often only interested in discovering a certain amount of patterns, and fine-tuning the *minsup* parameter is time-consuming. Depending on the choice of the *minsup* threshold, algorithms can become very slow and generate an extremely large amount of results or generate none or too few results, omitting valuable information. To address

this problem, it was proposed to redefine the problem of mining sequential patterns as the problem of mining the top-*k* sequential patterns, where k is the number of sequential patterns to be found and is set by the user. The current best algorithm for this problem is TSP [4]. However, in our experimental study, (cf. section 4), we found that it does not perform well on dense datasets. Therefore, an important research question is could we develop a more efficient algorithm for top-*k* sequential pattern mining than TSP? In this paper, we address this research question by proposing a novel algorithm named TKS (Top-K Sequential pattern mining). TKS is an efficient top-*k* algorithm for sequential pattern mining. It uses the same vertical database representation and basic candidate generation procedure as SPAM [3]. Moreover, TKS incorporates several efficient strategies to prune the search space and rely on a novel data structure named PMAP (Precedence Map) for avoiding costly bit vector intersection operations. An extensive experimental study with five real datasets shows that (1) TKS outperforms the state-of-the-art algorithm (TSP) by more than an order of magnitude in terms of execution time and memory usage. Moreover, we found that TKS has excellent performance on dense datasets.

The rest of the paper is organized as follows. Section 2 formally defines the problem of sequential pattern mining and top-*k* sequential pattern mining, and presents related work. Section 3 describes the TKS algorithm. Section 4 presents the experimental study. Finally, Section 5 presents the conclusion and discusses future work.

2 Problem Definition and Related Work

The problem of sequential pattern mining was proposed by Agrawal and Srikant [1] and is defined as follows. A *sequence database SDB* is a set of sequences $S = \{s_1, s_2..., s_s\}$ and a set of items $I = \{i_1, i_2, ..., i_m\}$ occurring in these sequences. An *item* is a symbolic value. An *itemset* $I = \{i_1, i_2, ..., i_m\}$ is an unordered set of distinct items. For example, the itemset $\{a, b, c\}$ represents the sets of items *a*, *b* and *c*. A *sequence* is an ordered list of itemsets $s = \langle I_1, I_2, ..., I_n \rangle$ such that $I_k \subseteq I$ for all $1 \leq k \leq n$. For example, consider the sequence database *SDB* depicted in Figure 1.a. It contains four sequences having respectively the *sequences ids* (SIDs) 1, 2, 3 and 4. In this example, each single letter represents an item. Items between curly brackets represent an itemset. For instance, the first sequence $\langle\{a, b\},\{c\},\{f\},\{g\},\{e\}\rangle$ indicates that items *a* and *b* occurred at the same time, were followed successively by *c*, *f*, *g* and lastly *e*. A sequence $s_a = \langle A_1, A_2, \ldots, A_n\rangle$ is said to be *contained in* another sequence $s_b = \langle B_1, B_2,\ldots, B_m\rangle$ if and only if there exists integers $1 \leq i_1 < i_2 < \ldots < i_n \leq m$ such that $A_1 \subseteq B_{i1}$, $A_2 \subseteq B_{i2}$, …, $A_n \subseteq B_{in}$ (denoted as $s_a \sqsubseteq s_b$). The *support of a subsequence* s_a in a sequence database *SDB* is defined as the number of sequences $s \in S$ such that $s_a \sqsubseteq s$ and is denoted by $sup(s_a)$. The *problem of mining sequential patterns* in a sequence database *SDB* is to find all frequent sequential patterns, i.e. each subsequence s_a such that $sup(s_a) \geq minsup$ for a threshold *minsup* set by the user. For example, Figure 1.b shows six of the 29 sequential patterns found in the database of Figure 1.a for *minsup* = 2. Several algorithms have been proposed for the problem sequential pattern mining such as PrefixSpan [2], SPAM [3], GSP and SPADE [9].

Problem definition. To address the difficulty of setting *minsup*, the problem of sequential pattern mining was redefined as the *problem of top-k sequential pattern min-*

ing [4]. It is to discover a set L containing k sequential patterns in a sequence database *SDB* such that for each pattern $s_a \in L$, there does not exist a sequential pattern $s_b \notin L \mid sup(s_b) > sup(s_a)$. For example, for the database of Figure 1.a and $k = 10$, the top-k sequential patterns are $\langle\{g\}\rangle$, $\langle\{a\},\{f\}\rangle$, $\langle\{a\}\rangle$, $\langle\{b\}, \{e\}\rangle$, $\langle\{b\}, \{g\}\rangle$, $\langle\{a\}, \{e\}\rangle$ and$\langle\{e\}\rangle$ with a support of 3, and $\langle\{b\}, \{f\}\rangle$, $\langle\{b\}\rangle$ and $\langle\{f\}\rangle$, with a support of 4. The definition of this problem is analogous to the definition of other top-k problems in the field of pattern mining such as top-k frequent itemset mining [1], top-k association rule mining [6] and top-k sequential rule mining.

The current state-of-the-art algorithm for top-k sequential pattern mining is TSP [4]. Two versions of TSP have been proposed for respectively mining (1) top-k sequential patterns and (2) top-k closed sequential patterns. In this paper, we are addressing the first case. Extending our algorithm to the second case will be considered in future work. The TSP algorithm is based on PrefixSpan [2]. TSP first generates frequent sequential patterns containing a single item. Then it recursively extends each pattern s by (1) projecting the database by s, (2) scanning the resulting projected database to identify items that appear more than *minsup* times after s, and (3) append these items to s. The main benefit of this projection-based approach is that it only considers patterns appearing in the database unlike "generate-and-test" algorithms [1, 4]. However, the downside of this approach is that projecting/scanning databases repeatedly is costly, and that cost becomes huge for dense databases where multiples projections have to be performed (c.f. experimental study presented in Section 4). Given this limitation, an important research challenge is to define an algorithm that would be more efficient than TSP and that would perform well on dense datasets.

SID	Sequences
1	$\langle\{a, b\},\{c\},\{f, g\},\{g\},\{e\}\rangle$
2	$\langle\{a, d\},\{c\},\{b\},\{a, b, e, f\}\rangle$
3	$\langle\{a\},\{b\},\{f\},\{e\}\rangle$
4	$\langle\{b\},\{f, g\}\rangle$

ID	Pattern	Supp.
p1	$\langle\{a\},\{f\}\rangle$	3
p2	$\langle\{a\},\{c\}\{f\}\rangle$	2
p3	$\langle\{b\},\{f,g\}\rangle$	2
p4	$\langle\{g\},\{e\}\rangle$	2
p5	$\langle\{c\},\{f\}\rangle$	2
p6…	$\langle\{b\}\rangle$	4

Fig. 1. A sequence database (left) and some sequential patterns found (right)

3 The TKS Algorithm

We address this research challenge by proposing a novel algorithm named TKS. TKS employs the vertical database representation and basic candidate-generation procedure of SPAM [2]. Furthermore, it also includes several efficient strategies to discover top-k sequential pattern efficiently. The next subsection reviews important concepts of the SPAM algorithm. Then, the following subsection describes the TKS algorithm.

3.1 The Database Representation and Candidate Generation Procedure

The *vertical database representation* [3] used in TKS is defined as follows. Let *SDB* be a sequence database containing q items and m sequences, where $size(i)$ denotes the number of itemsets in the i-th sequence of *SDB*. The *vertical database representation*

V(*SDB*) of *SDB* is defined as a set of *q* bit vectors of size $\sum_{i=1}^{m} size(i)$, such that each item *x* has a corresponding bit vector *bv(x)*. For each bit vector, the *j*-th bit represents the *p*-th itemset of the *t*-th sequence of *SDB*, such that $\sum_{i=1}^{\min(0,t-1)} size(i) < j < \sum_{i=1}^{t} size(i)$ and $p = j - \sum_{i=1}^{\min(0,t-1)} size(i)$. Each bit of a bit vector *bv(x)* is set to 1 if and only if *x* appears in the itemset represented by this bit, otherwise, it is set to 0. For example, the left part of Table 1 shows the bit vectors constructed for each item from the database of Figure 1.

Table 1. The vertical representation (left) and PMAP data structure (right)

item	bit vector
a	100001001100000
b	100000011010010
c	010000100000000
d	000001000000000
e	000010001000100
f	001000001001001
g	001100000000001

item	pairs of type <item, support>
a	<a,1, s> <b,2, s>, <b, 2, i>, <c, 2, s>, <d,1, i>, <e,2, s>, <e,1, i>, <f,2, s>, <f,1, i>, <g,1, s>
b	<a,1,s>, <b,1,s>,<c, 1,s>, <e,2,s>, <e,1,i>, <f, 4,s>, <f, 1,i>, <g,2,s>
c	<a,1,s>, <b,1,s>, <e,2,s>, <f,2,s>, <g,1,s>
d	<a,1,s>, <b,1,s>,<c,1,s>, <e,1,s>, <f,1,s>
e	<f,1,i>
f	<e,2,s>, <g,2,i>
g	<e,1,s>

The procedure for candidate generation [3] is presented in Figure 2. It takes as parameter a sequence database *SDB* and the *minsup* threshold. It first scans *SDB* once to construct *V*(*SDB*). At the same time, it counts the support of each single item. Then, for each frequent item *s*, it calls the procedure "SEARCH". This procedure outputs the pattern ⟨{s}⟩ and recursively explore candidate patterns starting with the prefix ⟨{s}⟩. The SEARCH procedure (cf. Figure 3) takes as parameters a sequential pattern *pat* and two sets of items to be appended to *pat* to generate candidates. The first set S_n represents items to be appended to *pat* by *s-extension*. The result of the *s-extension* of a sequential pattern $\langle I_1, I_2, \ldots, I_n \rangle$ with an item *x* is $\langle I_1, I_2, \ldots, I_n, \{x\} \rangle$ [3]. The second set S_n represents items to be appended to *pat* by *i-extension*. The result of the *i-extension* of a sequential pattern $\langle I_1, I_2, \ldots, I_n \rangle$ with an item *x* is $\langle I_1, I_2, \ldots, I_n \cup \{x\} \rangle$ [3]. For each candidate *pat'* generated by extension, SPAM calculates the candidate's bit vector *bv*(*pat'*) by performing a modified logical AND (see [3] for details) of the bit vectors associated to *pat* and the appended item. The support of the candidate is calculated without scanning *SDB* by counting the number of bits set to 1 representing distinct sequences in *bv*(*pat'*) [3]. If the pattern *pat*' is frequent, it is then used in a recursive call to SEARCH to generate patterns starting with the prefix *pat'*. Note that in the recursive call, only items that resulted in a frequent pattern by extension of *pat* will be considered for extending *pat'* (see [3] for justification). Moreover, note that infrequent patterns are not extended by the SEARCH procedure because of the Apriori property (any infrequent sequential pattern cannot be extended to form a frequent pattern) [3]. The candidate generation procedure is very efficient in dense datasets because performing the AND operation for calculating the support does not require scanning the original database unlike the projection-based approach of TSP, which in the worst case performs a database projection for each item appended to a pattern. However, there is a potential downside to the candidate generation procedure of SPAM. It is that it can generate candidates not occurring in the database. Therefore, it is not obvious that

building a top-*k* algorithm based on this procedure would result in an efficient algorithm.

SPAM(*SDB*, *minsup*)
1. Scan *SDB* to create *V*(*SDB*) and identify S_{init}, the list of frequent items.
2. **FOR** each item $s \in S_{init,}$
3. **SEARCH**($\langle s \rangle$, S_{init}, the set of items from S_{init} that are lexically larger than *s*, *minsup*).

Fig. 2. The SPAM algorithm

SEARCH(*pat*, $S_{n,}$ I_n, *minsup*)
1. Output pattern *pat.*
2. $S_{temp} := I_{temp} := \emptyset$
3. **FOR** each item $j \in S_{n,}$
4. **IF** the s-extension of *pat* is frequent **THEN** $S_{temp} := S_{temp} \cup \{i\}$.
5. **FOR** each item $j \in S_{temp,}$
6. **SEARCH**(the s-extension of *pat* with *j*, $S_{temp\ ,}$ elements in S_{temp} greater than *j*, *minsup*).
7. **FOR** each item $j \in I_{n,}$
8. **IF** the i-extension of *pat* is frequent **THEN** $I_{temp} := I_{temp} \cup \{i\}$.
9. **FOR** each item $j \in I_{temp,}$
10. **SEARCH**(i-extension of *pat* with *j*, $S_{temp\ ,}$ all elements in I_{temp} greater than *j*, *minsup*).

Fig. 3. The candidate generation procedure

3.2 The TKS Algorithm

We now present our novel top-*k* sequential pattern mining algorithm named TKS. It takes as parameters a sequence database *SDB* and *k*. It outputs the set of top-*k* sequential patterns contained in *SDB*.

Strategy 1. Raising Support Threshold. The basic idea of TKS is to modify the main procedure of the SPAM algorithm to transform it in a top-*k* algorithm. This is done as follows. To find the top-*k* sequential patterns, TKS first sets an internal *minsup* variable to 0. Then, TKS starts searching for sequential patterns by applying the candidate generation procedure. As soon as a pattern is found, it is added to a list of patterns *L* ordered by the support. This list is used to maintain the top-*k* patterns found until now. Once *k* valid patterns are found, the internal *minsup* variable is raised to the support of the pattern with the lowest support in *L*. Raising the *minsup* value is used to prune the search space when searching for more patterns. Thereafter, each time a frequent pattern is found, the pattern is inserted in *L*, the patterns in *L* not respecting *minsup* anymore are removed from *L*, and *minsup* is raised to the value of the least interesting pattern in *L*. TKS continues searching for more patterns until no pattern can be generated, which means that it has found the top-*k* sequential patterns. It can be easily seen that this algorithm is correct and complete given that the candidate generation procedure of SPAM is. However, in our test, an algorithm simply incorporating Strategy 1 does not have good performance.

TKS(*SDB, k*)
1. R := Ø. L := Ø. *minsup* := 0.
2. Scan *SDB* to create *V*(*SDB*).
3. Let S_{init} be the list of items in *V*(*SDB*).
4. **FOR** each item s ∈ $S_{init,}$ **IF** *s* is frequent according to bv(*s*) **THEN**
5. **SAVE**(*s*, *L*, *k*, *minsup*).
6. *R* := *R* ∪ {<*s*, $S_{init,}$. items from S_{init} that are lexically larger than *s*>}.
7. **WHILE** ∃ <*r*, S_1, S_2> ∈ *R* AND sup(*r*) ≥ *minsup* DO
8. Select the tuple <*r*, S_1, S_2> having the pattern *r* with the highest support in *R*.
9. **SEARCH**(*r*, S_1, S_2, *L*, *R*, *k*, *minsup*).
10. **REMOVE** <*r*, S_1, S_2> from R.
11. **REMOVE** from *R* all tuples <*r*, S_1, S_2> ∈ *R* | *sup*(*r*) < *minsup*.
12. **END WHILE**
13. **RETURN** *L*.

Fig. 4. The TKS algorithm

SEARCH(*pat*, $S_{n,}$ $I_{n,}$ *L*, *R*, *k*, *minsup*)
1. S_{temp} := I_{temp} := ∅
2. **FOR** each item *j* ∈ $S_{n,}$
3. **IF** the s-extension of *pat* is frequent **THEN** S_{temp} := S_{temp} ∪{*i*}.
4. **FOR** each item *j*∈ $S_{temp,}$
5. **SAVE**(s-extension of *pat* with *j*, *L*, *k*, *minsup*).
6. *R* := *R* ∪ {< s-extension of *pat* with *j*, S_{temp}, all elements in S_{temp} greater than *j*>}.
7. **FOR** each item *j* ∈ I_{n},
8. **IF** the i-extension of *pat* is frequent **THEN** I_{temp} := I_{temp} ∪{*i*}.
9. **FOR** each item *j* ∈ $I_{temp,}$
10. **SAVE**(i-extension of *pat* with *j*, *L*, *k*, *minsup*).
11. *R* := *R* ∪ {< the s-extension of *pat* with *j*, S_{temp} , all elements in I_{temp} greater than *j* >}.

Fig. 5. The modified candidate generation procedure

SAVE(*r*, *L*, *k*, *minsup*)
1. *L* := *L*∪{*r*}.
2. **IF** |*L*| >*k* **THEN**
3. **IF** *sup*(*r*) > *minsup* **THEN**
4. **WHILE** |*L*| > *k* AND ∃*s* ∈ *L*| *sup*(*s*) = *minsup*, **REMOVE** *s* from *L*.
5. **END IF**
6. Set *minsup* to the lowest support of patterns in *L*.
7. **END IF**

Fig. 6. The SAVE procedure

Strategy 2. Extending the Most Promising Patterns. To improve the performance of TKS, we have added a second strategy. It is to try to generate the most promising sequential patterns first. The rationale of this strategy is that if patterns with high support are found earlier, it allows TKS to raise its internal *minsup* variable faster, and thus to prune a larger part of the search space. To implement this strategy, TKS uses an internal variable *R* to maintain at any time the set of patterns that can be extended

to generate candidates. TKS then always extends the pattern having the highest support first.

The pseudocode of the TKS version incorporating Strategy 1 and Strategy 2 is shown in Figure 4. The algorithm first initializes the variables *R* and *L* as the empty set, and *minsup* to 0 (line *1*). Then, *SDB* is scanned to create *V*(*SDB*) (line *2*). At the same time, a list of all items in *SDB* is created (S_{init}) (line *3*). For each item *s*, its support is calculated based on its bit vector *bv*(*s*) in *V*(*SDB*). If the item is frequent, the SAVE procedure is called with ⟨s⟩ and *L* as parameters to record ⟨s⟩ in *L* (line *4* and *5*). Moreover, the tuple <*s*, S_{init}, items from S_{init} that are lexically larger than *s*> is saved into *R* to indicate that ⟨s⟩ can be extended to generate candidates (line *6*). After that, a WHILE loop is performed. It recursively selects the tuple representing the pattern *r* with the highest support in *R* such that *sup*(*r*) ≥ *minsup* (line *7* and *8*). Then the algorithm uses the tuple to generate patterns by using the SEARCH procedure depicted in Figure 5 (line *9*). After that, the tuple is removed from *R* (line *10*), as well as all tuples for patterns that have become infrequent (line *11*). The idea of the WHILE loop is to always extend the rule having the highest support first because it is more likely to generate rules having a high support and thus to allow to raise *minsup* more quickly for pruning the search space. The loop terminates when there is no more pattern in *R* with a support higher than *minsup*. At this moment, the set *L* contains the top-*k* sequential patterns (line *13*).

The SAVE procedure is shown in Figure 6. Its role is to raise *minsup* and update the list *L* when a new frequent pattern *r* is found. The first step of SAVE is to add the pattern *r* to *L* (line *1*). Then, if *L* contains more than *k* patterns and the support is higher than *minsup*, patterns from *L* that have exactly the support equal to *minsup* can be removed until only *k* rules are kept (line *3 to 5*). Finally, *minsup* is raised to the support of the rule in *L* having the lowest support. (line *6*). By this simple scheme, the top-k rules found are maintained in *L*.

Note that to improve the performance of TKS, in our implementation, sets *L* and *R* are implemented with data structures supporting efficient insertion, deletion and finding the smallest/largest element. In our implementation, we used a Fibonacci heap for *L* and *R*. It has an amortized time cost of *O*(1) for insertion and obtaining the minimum/maximum, and *O*(*log(n)*) for deletion [9].

Strategy 3. Discarding Infrequent Items in Candidate Generation. This strategy improves TKS' execution time by reducing the number of bit vector intersections performed by the SEARCH procedure. The motivation behind this strategy is that we found that a major cost of candidate generation is performing bit vector intersections because bit vectors can be very long for large datasets. This strategy is implemented in two phases. First, TKS records in a hash table the list of items that become infrequent when *minsup* is raised by the algorithm. This is performed in line 6 of the SAVE procedure by replacing "**REMOVE** *s* from *L*." by "**REMOVE** s from *L* and **IF** *s* contains a single item **THEN** register it in the hash map of discarded items". Note that it is not necessary to record infrequent items discovered during the creation of *V*(*SDB*) because those items are not considered for pattern extension.

Second, each time that the SEARCH procedure considers extending a sequential pattern *pat* with an item *x* (by *s-extension* or *i-extension)*, the item is skipped if it is contained in the hash table. Skipping infrequent items allows avoiding performing costly bit vector intersections for these items. Integrating this strategy does not affect the output of the algorithm because appending an infrequent item to a sequential pattern cannot generate a frequent sequential pattern.

Strategy 4. Candidate Pruning with Precedence Map. We have integrated a second strategy in TKS to reduce more aggressively the number of bit vector intersections. This strategy requires building a novel structure that we name *Precedence Map (PMAP).* This structure is built with a single database scan over *SDB*. The PMAP structure indicates for each item *i*, a list of triples of the form <*j, m, x*> where *m* is an integer representing the number of sequences where *j* appears after *i* in *SDB* by *x-extension* ($x \in \{i, s\}$). Formally, an item *i* is said to *appear after an item j* by *s*-extension in a sequence $\langle A_1, A_2, \ldots, A_n \rangle$ if $j \in A_x$ and $i \in A_y$ for integers *x* and *y* such that $1 \leq x < y \leq n$. An item *i* is said to *appear after an item j* by *i*-extension in a sequence $\langle A_1, A_2, \ldots, A_n \rangle$ if $i, j \in A_x$ for an integer *x* such that $1 \leq x \leq n$ and *j* is lexicographically greater than *i*. For example, the PMAP structure built for the sequence database of Figure 1 is shown in right part of Table 1. In this example, the item *f* is associated with the pair <*e*, 2, *s*> because *e* appears after *f* by *s*-extension in two sequences. Moreover, *f* is associated with the pair <*g*, 2, *i*> because *g* appears after *f* by *i-extension* in two sequences. To implement PMAP, we first considered using two matrix (one for *i-extensions* and one for *s-extension*). However, for sparse datasets, several entries would be empty, thus potentially wasting large amount of memory. For this reason, we instead implemented PMAP as a hash table of hash sets. Another key implementation decision for PMAP is when the structure should be built. Intuitively, one could think that constructing PMAP should be done during the first database scan at the same time as *V*(*SDB*) is constructed. However, to reduce the size of PMAP, it is better to build it in a second database scan so that infrequent items can be excluded from PMAP during its construction. The PMAP structure is used in the SEARCH procedure, which we modified as follows. Let a sequential pattern *pat* being considered for *s-extension* (equivalently *i-extension)* with an item *x*. If there exists an item *a* in *pat* associated to an entry <*x, m, s*> *in* PMAP (equivalently <*x, m, i*>) and *m* < *minsup*, then the pattern resulting from the extension of *pat* with *x* will be infrequent and thus the bit vector intersection of *x* with *pat* does not need to be done. It can be easily seen that this pruning strategy does not affect the algorithm output, since if an item *x* does not appear more than *minsup* times after an item *y* from a pattern *pat*, any pattern containing *y* followed by *x* will be infrequent. Furthermore, *x* can be removed from S_{temp} (equivalently I_{temp}).

Optimizations of the bit vector representation. Beside the novel strategies that we have introduced, optimizations can be done to optimize the bit vector representation and operations. For example, bit vectors can be compressed if they contain contiguous zeros and it is possible to remember the first and last positions of bits set to 1 in each

bit vector to reduce the cost of intersection. These optimizations are not discussed in more details here due to the space limitation.

Correctness and completeness of the algorithm. Since SPAM is correct and complete and Strategy 2, 3 and 4 have no influence on the output (only parts of the search space that lead to infrequent patterns is pruned), it can be concluded that TKS is correct and complete.

4 Experimental Study

We performed multiple experiments to assess the performance of the TKS algorithm. Experiments were performed on a computer with a third generation Core i5 processor running Windows 7 and 1 GB of free RAM. We compared the performance of TKS with TSP, the state-of-the-art algorithm for top-*k* sequential pattern mining. All algorithms were implemented in Java. The source code of all algorithms and datasets can be downloaded as part of the SPMF data mining framework (http://goo.gl/hDtdt). All memory measurements were done using the Java API. Experiments were carried on five real-life datasets having varied characteristics and representing four different types of data (web click stream, text from books, sign language utterances and protein sequences). Those datasets are *FIFA*, *Leviathan*, *Bible*, *Sign* and *Snake*. Table 1 summarizes their characteristics.

Table 1. Datasets' Characteristics

dataset	**sequence count**	**distinct item count**	**avg. seq. length (items)**	**type of data**
Leviathan	5834	9025	33.81 (std= 18.6)	book
Bible	36369	13905	21.64 (std = 12.2)	book
Sign	730	267	51.99 (std = 12.3)	sign language utterances
Snake	163	20	60 (std = 0.59)	protein sequences
FIFA	20450	2990	34.74 (std = 24.08)	web click stream

Experiment 1. Influence of the *k* parameter. We first ran TKS and TSP on each dataset while varying *k* from 200 to 3000 (typical values for a top-*k* pattern mining algorithm) to assess the influence of *k* on the execution time and the memory usage of the algorithms. Results for *k* =1000, 2000 and 3000 are shown in Table 2. As it can be seen in this table, TKS largely outperforms TSP on all datasets in terms of execution time and memory usage. TKS can be more than an order of magnitude faster than TSP and use up to an order of magnitude less memory than TSP. Note that no result are given for TSP for the *FIFA* dataset when *k* = 2000 and *k* = 3000 because it run out of memory. Figure 2 shows detailed results for the *Bible* and *Snake* dataset. From this figure, it can be seen that TKS has better scalability than TSP with respect to *k* (detailed results are similar for other datasets and not shown due to page limitation). From this experiment, we can also observe that TKS performs very well on dense

datasets. For example, on the very dense dataset *Snake*, TKS uses 13 times less memory than TSP and is about 25 times faster for k = 3000.

Table 2: Results for k = 1000, 2000 and 3000

Dataset	**Algorithm**	**Execution Time (s)**			**Maximum Memory Usage (MB)**		
		***k*=1000**	***k*=2000**	***k*=3000**	***k*=1000**	***k*=2000**	***k*=3000**
Leviathan	**TKS**	**10**	**23**	**38**	**302**	**424**	**569**
	TSP	103	191	569	663	856	974
Bible	**TKS**	**16**	**43**	**65**	**321**	**531**	**658**
	TSP	88	580	227	601	792	957
Sign	**TKS**	**0.5**	**0.8**	**1.2**	**46**	**92**	**134**
	TSP	4.8	7.6	9.1	353	368	383
Snake	**TKS**	**1.1**	**1.63**	**1.8**	**19**	**38**	**44**
	TSP	19	33	55	446	595	747
FIFA	**TKS**	**15**	**34**	**95**	**436**	**663**	**796**
	TSP	182	O.O.M.	O.O.M.	979	O.O.M.	O.O.M.

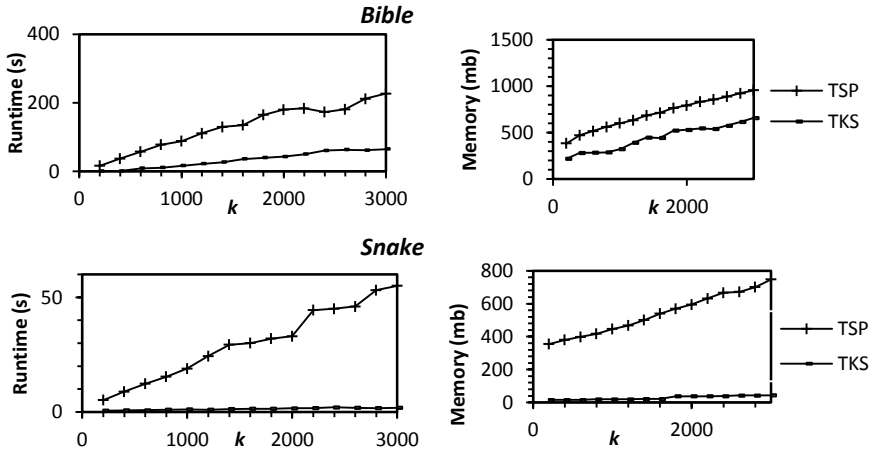

Fig. 7. Results of varying k for the *Bible* and *Snake* datasets

Experiment 2. Influence of the strategies. We next evaluated the benefit of using strategies for reducing the number of bit vector intersections in TKS. To do this, we compared TKS with a version of TKS without Strategy 4 (TKS W4) and without both Strategy 3 and Strategy 4 (TKS W3W4). We varied k from 200 to 3000 and measured the execution time and number of bit vector intersection performed by each version of TKS. For example, the results for the *Sign* dataset are shown in Figure 8. Results for other datasets are similar and are not shown due to space limitation. As it can be seen on the left side of Figure 8, TKS outperforms TKS W4 and TKS W3W4 in execution time by a wide margin. Moreover, as it can be seen on the right side of Figure 8,

Strategy 3 and Strategy 4 are effective strategies that greatly reduce the number of bit vector intersections. Note that we also considered the case of removing without Strategy 2. However, the resulting algorithm would not terminate on most datasets. For this reason, results are not shown.

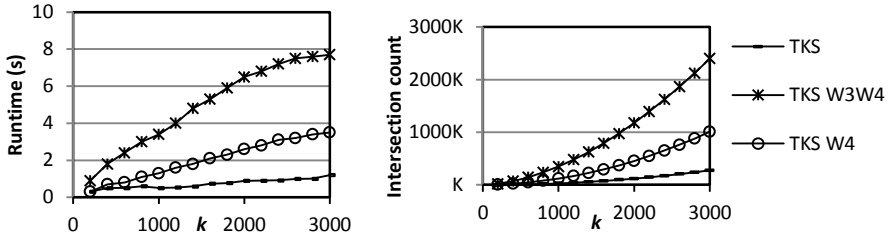

Fig. 8. Influence of the strategies for the *Sign* dataset

Experiment 3. Influence of the number of sequences. We also ran TKS and TSP on the five datasets while varying the number of sequences in each dataset to assess the scalability of TKS and TSP. For this experiment, we used k=1000, and varied the database size from 10% to 100 % of the sequences in each dataset. Results are shown in Figure 9 for the *Leviathan* dataset, which provides representative results. We found that both TKS and TSP shown excellent scalability.

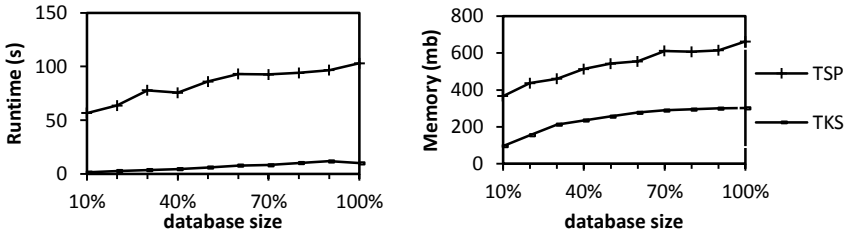

Fig. 9. Influence of the number of sequences for the *Leviathan* dataset

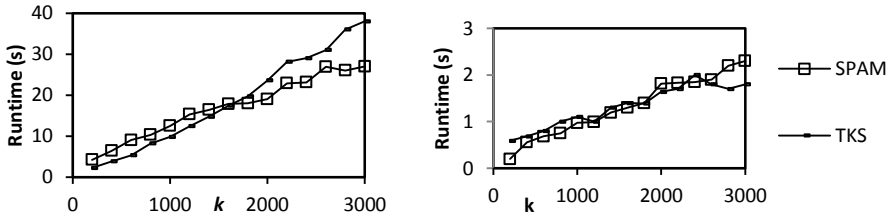

Fig. 10. Comparison of SPAM and TKS runtime for *Leviathan* (left) and *Snake* (right)

Experiment 4. Performance comparison with SPAM. To further assess the efficiency of TKS, we compared its performance with SPAM for the scenario where the user would tune SPAM with the optimal minimum support threshold to generate *k* patterns (which is very hard for a user, in practice). The results for the *Leviathan* and

Snake dataset are shown in Figure 10. Results for other datasets are not shown due to space limitation but are similar. We conclude from the results that the execution time of TKS is very close to that of SPAM for the optimal support and that TKS shows similar scalability to SPAM. This is excellent because top-*k* sequential pattern mining is a much harder problem than sequential pattern mining since *minsup* has to be raised dynamically, starting from 0 [4, 6, 7, 10],

5 Conclusion

We proposed TKS, an algorithm to discover the *top-k* sequential patterns having the highest support, where *k* is set by the user. To generate patterns, TKS relies on a set of efficient strategies and optimizations that enhance its performance. An extensive experimental study show that TKS (1) outperforms TSP, the state-of the art algorithm by more than an order of magnitude in terms of execution time and memory usage, (2) has better scalability with respect to *k* and (3) has a very low performance overhead compared to SPAM. The source code of TKS as well as all the datasets and algorithms used in the experiment can be downloaded from http://goo.gl/hDtdt.

References

1. Agrawal, R. and Srikant, R.: Mining Sequential Patterns, Proc. Int. Conf. on Data Engineering, pp. 3-14 (1995)
2. Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U., Hsu, M.: Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach, IEEE Trans. Knowledge and Data Engineering, vol. 16, no. 10, pp. 1-17 (2001)
3. Ayres, J., Flannick, J., Gehrke, J. and Yiu, T.: Sequential PAttern mining using a bitmap representation, Proc. 8th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD 2002), July 23-26, 2002, Edmonton, Alberta, pp. 429-435 (2002)
4. Tzvetkov, P. Yan, X. and Han, J.: TSP: Mining Top-k Closed Sequential Patterns, Knowledge and Information Systems, vol. 7, no. 4, pp. 438-457 (2005)
5. Mabroukeh, N. R. and Ezeife, C. I.: A taxonomy of sequential pattern mining algorithms, ACM Computing Surveys, vol. 43, no. 1, pp. 1-41 (2010)
6. Fournier-Viger, P., Wu, C.-W., Tseng, V. S.: Mining Top-K Association Rules, Proc. of the 25th Canadian Conf. on Artificial Intelligence (AI 2012), Springer, pp. 61-73 (2012)
7. Kun Ta, C., Huang, J.-L., and Chen, M.-S.: Mining Top-k Frequent Patterns in the Presence of the Memory Constraint, VLDB Journal, vol. 17, no. 5, pp. 1321-1344 (2008)
8. Han, J. and Kamber, M.: Data Mining: Concepts and Techniques, 2nd ed., San Francisco:Morgan Kaufmann Publ. (2006)
9. Cormen, T. H., Leiserson, C. E., Rivest, R. and Stein, C.: Introduction to Algorithms, 3rd ed., Cambridge:MIT Press (2009)
10. Fournier-Viger, P. and Tseng, V. S.: Mining Top-K Sequential Rules, Proc. of the 7th Intern. Conf. on Advanced Data Mining and Applications (ADMA 2011), Springer LNAI 7121, pp.180-194 (2011)