Mining Maximal Sequential Patterns without Candidate Maintenance

Philippe Fournier-Viger[1], Cheng-Wei Wu[2] and Vincent S. Tseng[2]

[1]Departement of Computer Science, University of Moncton, Canada
[2]Dep. of Computer Science and Information Engineering, National Cheng Kung University

philippe.fournier-viger@umoncton.ca,
silvemoonfox@hotmail.com,tsengsm@mail.ncku.edu.tw

Abstract. Sequential pattern mining is an important data mining task with wide applications. However, it may present too many sequential patterns to users, which degrades the performance of the mining task in terms of execution time and memory requirement, and makes it difficult for users to comprehend the results. The problem becomes worse when dealing with dense or long sequences. As a solution, several studies were performed on mining maximal sequential patterns. However, previous algorithms are not memory efficient since they need to maintain a large amount of intermediate candidates in main memory during the mining process. To address these problems, we present a both time and memory efficient algorithm to efficiently mine maximal sequential patterns, named MaxSP (Maximal Sequential Pattern miner), which computes all maximal sequential patterns without storing intermediate candidates in main memory. Experimental results on real datasets show that MaxSP serves as an efficient solution for mining maximal sequential patterns.

Keywords: sequences, sequential pattern mining, compact representation, maximal sequential patterns

1 Introduction

Mining useful patterns in sequential data is a challenging task in data mining. Many studies have been proposed for mining interesting patterns in sequence databases [1, 2, 3] *Sequential pattern mining* is probably the most popular research topic among them. A *sub-sequence* is called *sequential pattern* or *frequent sequence* if it frequently appears in a sequence database, and its frequency is no less than a user-specified *minimum support threshold minsup*. Sequential pattern mining plays an important role in data mining and is essential to a wide range of applications such as the *analysis of web click-streams*, *program executions*, *medical data*, *biological data* and *e-learning data* [1, 2, 3]. Several algorithms have been proposed for sequential pattern mining such as SPAM [4], SPADE [5] and PrefixSpan [6]. However, a drawback of these algorithms is that they may present too many sequential patterns to users. A very large number of sequential patterns makes it difficult for users to analyze results to gain insightful knowledge. It may also cause the algorithms to become inefficient in terms of time and memory because the more sequential patterns the algorithms produce, the more resources they consume. The problem becomes worse when the database contains long sequential patterns. For example, consider a sequence database containing a sequential pattern having 20 distinct items. A sequential pattern mining algorithm will present the sequential pattern as well as its 2^{20-1} subsequences to users. This will most likely make the algorithm fail to terminate in reasonable time and run out of storage space. For example, the well-known sequential pattern mining algorithm PrefixSpan would have to perform 2^{20} database projection operations to produce the results.

To reduce the computational cost of the mining task and present fewer but representative patterns to users, many studies focused on developing concise representations of sequential patterns. One of the representations that has been proposed is *closed sequential patterns* [7, 8, 9]. A closed sequential pattern is a sequential pattern that is not strictly included in another pattern having the same frequency. Several approaches have been proposed for mining closed sequential patterns in sequence databases such as BIDE [7], CloSpan [8] and ClaSP [9]. Although these algorithms mines a compact set of sequential patterns, the set of closed patterns is still too large for dense databases or database containing long sequences.

To address this problem, it was proposed to mine *maximal sequential patterns* [10, 11, 12, 13, 14]. A maximal sequential pattern is a closed pattern that is not strictly included in another closed pattern. The set of maximal sequential patterns is thus generally a very small subset of the set of (closed) sequential patterns. It is widely recognized that mining maximal patterns can be faster than mining all (closed) patterns. Besides, the set of maximal sequential patterns is representative because all sequential patterns can be derived from it. Furthermore, the exact frequency of the sequential patterns can be obtained with a single database pass. This method thus provides an alternative solution to find all sequential patterns when the traditional algorithms cannot successfully mine (closed) sequential patterns in the databases.

Maximal sequential pattern mining is important and has been adopted for many applications such as discovering frequent longest common subsequences in a text, analysis of DNA sequences, data compression and web log mining [10].

Although maximal sequential pattern mining is desirable and useful in many applications, it is still a challenging data mining task that has not been deeply explored. Only few algorithms have been proposed for efficiently mining maximal sequential patterns. MSPX [12] is an approximate algorithm and therefore it provides an incomplete set of maximal patterns to the user, and thus may omit important information. DIMASP algorithm [10] is designed for the special case where sequences are strings (no more than an item can appear at the same time) and where no pair of contiguous items appears more than once in each sequence. AprioriAdjust algorithm [13] is an apriori-like algorithm, which may suffer from the drawbacks of the candidate generation-and-test paradigm. In other word, it may generate a large number of candidate patterns that do not appear in the input database and require to scan the original database many times. MSPX [12] and MFSPAN [14] algorithms need to maintain a large amount of intermediate candidates in main memory during the mining process. Although the above algorithms are pioneers for maximal sequential pattern mining, they are not memory efficient since they need to maintain a large amount of intermediate candidates in main memory during the mining process [10, 11, 12, 13, 14].

To address the above issues, we propose a both time and memory efficient algorithm, named *MaxSP* (*Maximal Sequential Pattern miner*), to efficiently mine maximal sequential patterns in sequence databases. The proposed algorithm is developed for the general case of a sequence database rather than strings and it can capture the complete set of maximal sequential patterns with only two database scans. Moreover, it discovers all maximal sequential patterns neither producing redundant candidates nor storing intermediate candidates in main memory. Whenever a maximal pattern is discovered by MaxSP, it can be outputted immediately. We performed an experimental study with five real-life datasets to evaluate the performance of MaxSP. We compared its performance with the BIDE algorithm [8], one of the current best algorithms for mining closed sequential patterns without storing intermediate candidates in memory. Results show that MaxSP outperforms BIDE in terms of execution and memory consumption and that the set of maximal patterns is much smaller than the set of closed patterns.

The rest of the paper is organized as follows. Section 2 formally defines the problem of maximal sequential pattern mining and its relationship to sequential pattern mining. Section 3 describes the MaxSP algorithm. Section 4 presents the experimental study. Finally, Section 5 presents the conclusion and future work.

2 Problem Definition

The problem of sequential pattern mining was proposed by Agrawal and Srikant [1]. 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 $I_k \subseteq I$ $(1 \leq k \leq n)$. For example, consider the sequence database depicted in Figure 1. 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 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 $s_a = \langle A_1, A_2, …, A_n\rangle$ is *contained in* another sequence $s_b = \langle B_1, B_2,…, B_m\rangle$ iff there exists integers $1 \leq i_1 < i_2 < … < i_n \leq m$ 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 $sup(s_a)$.

Definition 1. 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 2 shows the 29 sequential patterns found in the database of Figure 1 for $minsup = 2$.
Several algorithms have been proposed to mine sequential pattern such as SPAM [4], PrefixSpan [5], and SPADE [6]. To reduce the number of sequential patterns found

and find representative patterns, it was proposed to mine *closed* and *maximal sequential patterns*.

Definition 2. A *closed sequential pattern is* a frequent sequential pattern that is not strictly included in another sequential pattern having the same support [8, 9].

Definition 3. A *maximal sequential pattern* is a frequent sequential pattern that is not strictly included in another frequent sequential pattern. An equivalent definition is that a sequential pattern is maximal if it is a closed sequential pattern that is not strictly included in another closed sequential pattern.

Property 1. It can be easily seen that maximal patterns are a subset of the set of closed patterns and that closed patterns are a subset of the set of frequent sequential patterns. **Rationale**. This follows directly from the above definitions. **Example.** Consider the database of Figure 1 and *minsup* = 2. There are 29 sequential patterns (shown in Figure 2), such that 15 are closed (identified by the letter 'C') and only 10 are maximal (identified by the letter 'M').

Property 2. Maximal patterns are a lossless representation of all frequent sequential patterns (they allow recovering all frequent sequential patterns). **Proof.** By definition, a maximal sequential pattern has no proper super-sequence that is a frequent sequential pattern. Thus, if a pattern is frequent, it is either a proper subsequence of a maximal pattern or a maximal pattern. Figure 3 presents a simple algorithm for recovering all sequential patterns from the set of maximal sequential patterns. It generates all the subsequences of all the maximal patterns. Furthermore, a check is performed to detect if a sequential patterns has already been output (line 3) because a sequential pattern may be a subsequence of more than one maximal pattern.

3 The MaxSP Algorithm

The MaxSP algorithm is a pattern-growth algorithm inspired by PrefixSpan [5]. We therefore first briefly introduce the PrefixSpan algorithm. Then, we present the MaxSP algorithm and discuss optimizations.

3.1 Discovering sequential patterns by pattern-growth

PrefixSpan [5] is one of the most efficient sequential pattern mining algorithm. It offers several interesting properties such as being a pattern-growth approach, i.e. it discovers patterns directly without generating candidates. The pseudocode of the main steps of PrefixSpan is shown in Figure 4. PrefixSpan takes three parameters as input. The first parameter is a sequence database *SDB*. The second parameter is the user-defined minimum support threshold *minsup*. The third parameter is a prefix sequence *P*, which is initially set to the empty sequence ⟨⟩. The output of PrefixSpan is the set of frequent sequential patterns. PrefixSpan operates as follows. It first scans *SDB* once to calculate the support of each single item (line 1). Then, for each frequent item *i*, the algorithm outputs the sequential pattern ⟨{*i*}⟩ (line 2-4). For each frequent item *i*, a projection of the database *SDB* by *i* is performed to obtain a projected database SDB_i (line 5). Then, a recursive call is performed with parameters SDB_i, *minsup* and the concatenation of the prefix *P* with {*i*} (line 6). The recursive call will then consider extending the pattern ⟨{*i*}⟩ with single items to form larger patterns. By the means of the recursive calls, the PrefixSpan algorithm recursively appends items one item at a time to discover larger patterns. The database projection operation is performed as follows.

SID	Sequences
1	⟨{*a*, *b*},{*c*},{*f*, *g*},{*g*},{*e*}⟩
2	⟨{*a*, *d*},{*c*},{*b*},{*a*, *b*, *e*, *f*}⟩
3	`<{a},{b},{f,g},{e}>`
4	⟨{*b*},{*f*, *g*}⟩

Fig. 1. A sequence database

Pattern	Sup.	Pattern	Sup.
⟨{*a*}⟩	3⟩ ⟩ C	⟨{*b*},{*g*},{*e*}⟩	2 CM
⟨{*a*},{*g*}⟩	2 ⟩	⟨{*b*},{*f*}⟩	4 C ⟩
⟨{*a*},{*g*},{*e*}⟩	2⟩ CM	⟨{*b*},{*f*, *g*}⟩	2 CM
⟨{*a*},{*f*}⟩	3⟩ C	⟨{*b*},{*f*},{*e*}⟩	2 CM
⟨{*a*},{*f*},{*e*}⟩	2⟩ CM	⟨{*b*},{*e*}⟩	3 C
⟨{*a*},{*c*}⟩	2⟩	⟨{*c*}⟩	2⟩
⟨{*a*},{*c*},{*f*}⟩	2 CM	⟨{*c*},{*f*}⟩	2⟩
⟨{*a*},{*c*},{*e*}⟩	2 CM	⟨{*c*},{*e*}⟩	2⟩
⟨{*a*},{*b*}⟩	2⟩	⟨{*e*}⟩	3⟩
⟨{*a*},{*b*},{*f*}⟩	2⟩ CM	⟨{*f*}⟩	4⟩
⟨{*a*},{*b*},{*e*}⟩	2⟩ CM	⟨{*f*, *g*}⟩	2⟩
⟨{*a*},{*e*}⟩	3⟩ C	⟨{*f*},{*e*}⟩	2⟩
⟨{*a*, *b*}⟩	2⟩ CM	⟨{*g*}⟩	3⟩
⟨{*b*}⟩	4⟩	⟨{*g*},{*e*}⟩	2⟩
⟨{*b*},{*g*}⟩	3⟩ C ⟩		

C = Closed M = Maximal

Fig. 2. Sequential patterns found for *minsup* = 2 (right)

```
RECOVERY (a set of maximal patterns M)
1.  FOR each sequential pattern j ∈ M,
2.      FOR each subsequence k of j,
3.          IF k has not been output
4.              THEN output k.
```

Fig. 3. Algorithm to recover all frequent sequential patterns from maximal patterns

Definition 4. The *projection of a sequence database SDB by a prefix P* is the projection of each sequence from *SDB* containing *P* by the prefix *P*.

Definition 5. The *projection of a sequence S by a prefix P* is the part of the sequence occurring immediately after the first occurrence of the prefix *P* in the sequence *S*. For instance, the projection of ⟨{*a*},{*c*},{*a*},{*e*}⟩ by the item *a* is the sequence ⟨{*c*},{*a*},{*e*}⟩ and the projection of ⟨{*a*},{*c*},{*b*},{*e*}⟩ by the prefix ⟨{*c*},{*b*}⟩ is ⟨{*e*}⟩.

Note that performing a database projection does not require to make a physical copy of the database. For memory efficiency, a projected database is rather represented by a set of pointers on the original database (this optimization is called *pseudo-projection*) [5]. Also, note that the pseudo-code presented in Figure 4 is simplified. The actual PrefixSpan algorithm needs to consider that an item can be appended to the current prefix *P* by *i-extension* or *s-extension* when counting the support of single items. An *i-extension* is to append an item to the last itemset of prefix *P*. An *s*-extension is to append an item as a new itemset after the last itemset of prefix *P*. The interested reader is referred to [5] for more details. The PrefixSpan algorithm is correct and complete. It enumerates all frequent sequential patterns thanks to the anti-monotonicity property, which states that the support of a proper supersequence *X* of a sequential pattern *S* can only be lower or equal to the support of *S* [2]. PrefixSpan is said to discover sequential patterns without candidate generation because only frequent items are concatenated with the current prefix *P* to generate patterns at each recursive call of the algorithm.

```
PrefixSpan (a sequence database SDB, a threshold minsup, a prefix P initially set to ⟨⟩)
1.  Scan SDB once to count the support of each item.
2.  FOR each item i with a support ≥ minsup
3.      P' := Concatenate(P, i).
4.      Output the pattern P'.
5.      SDBi := DatabaseProjection(SDB, i ).
6.      PrefixSpan(SDB_i, minsup, P').
```

Fig. 4. The main steps of the PrefixSpan algorithm

3.2 The MaxSP algorithm

The main challenge to design a maximal sequential pattern mining algorithm based on PrefixSpan is how to determine if a given frequent sequential pattern generated by the PrefixSpan is maximal. A naïve approach would be to keep all frequent sequential patterns found until now into memory. Then, every time that a new frequent sequential pattern would be found, the algorithm would compare the pattern with previously found patterns to determine if (1) the new pattern is included in a previously found pattern or (2) if some previously found pattern(s) are included in the new pattern. The first case would indicate that the new pattern is not maximal. The second case would indicate that some previously found pattern(s) are not maximal. This approach is used for example in CloSpan [8] for closed sequential pattern mining. The drawback of this approach is that it can consume a large amount of memory if the number of patterns is large, and it is becomes very time consuming if a very large number of patterns is found, because a very large number of comparisons would have to be performed [7]. In this paper, we present a new checking mechanism that can determine if a pattern is maximal without having to compare a new pattern with previously found patterns. The mechanism is inspired by the mechanism used in the BIDE algorithm for checking if a pattern is closed [7]. In this subsection, we first introduce important definitions and then we present our solution. Note that in the following, we use sequences where itemsets contain single items (strings) for the sake of simplicity. Nevertheless, the definitions that we present can be easily extended for the general case of a sequence of itemsets containing multiple items (our implementation handle the general case of itemsets).

Definition 4. Let be a prefix *P* and a sequence *S* containing *P*. The *first instance of the prefix P in S* is the subsequence of *S* starting from the first item in *S* until the end of the first instance of *P* in *S*. For example, the first instance of ⟨{a},{b}⟩ in ⟨{a},{a},{b},{e}⟩ is ⟨{a},{a}, {b}⟩.

Definition 5. Let be a prefix *P* and a sequence *S* containing *P*. The *last instance of the prefix P in S* is the subsequence of *S* starting from the first item in *S* until the end of the first instance of *P* in *S*. For example, the last instance of ⟨{a},{b}⟩ in ⟨{a},{b},{b},{e}⟩ is ⟨{a},{b}, {b}⟩.

Definition 6. Let be a prefix *P* containing *n* items and a sequence *S* containing *P*. The *i-th last-in-last appearance of P* in *S* is denoted as LL_i and defined as follows. If $i = n$, it is the last appearance of the *i*-th item of *P* in the last instance of *P* in *S*. If $1 < i < n$, it is the last appearance of the *i*-th item of *P* in the last instance of *P* in *S* such that LL_i must appear before LL_{i+1}. For example, the first last-in-last appearance of ⟨{*a*},{*c*}⟩ in the sequence ⟨{*a*},{*a*},{*c*},{*e*},{*c*}⟩ is the second *a*, while the second last-in-last appearance in the sequence ⟨{*a*},{*a*},{*c*},{*e*},{*c*}⟩ is the second *c*.

Definition 7. Let be a prefix *P* containing *n* items and a sequence *S* containing *P*. Let P_{n-1} denotes the first *n* items of *P*. The *i-th maximum period of P in S* is defined as follows. If $1 < i < n$, it is the piece of sequence between the end of the first instance of P_{n-1} in *S* and the *i*-th last-in-last appearance of *P* in *S*. If $i = 1$, it is the piece of sequence in *S* before the first last-in-last appearance of *P* in *S*. For example, consider *S* = ⟨{*a*},{*b*},{*c*},{*d*}⟩ and *P* = ⟨{*a*},{*b*}⟩. The first maximum period of *P* in *S* is the empty sequence, while the second maximum period is ⟨{*b*},{*c*}⟩.

Based upon the above definitions, we propose a mechanism to determine if a frequent pattern *P* is maximal without maintaining previously found patterns in memory. The mechanism consists of verifying if *P* can be extended by appending items to form a larger frequent sequential pattern. If yes, it indicates that the pattern *P* is not maximal (by definition 3, a maximal pattern has no proper supersequence that is frequent). Otherwise, *P* is maximal and can be immediately output. The mechanism is implemented by two separate checks, which we respectively name *maximal-backward-extension check* and *maximal-forward-extension check*, and are defined as follows.

Definition 8. Let be a frequent sequential pattern *P* containing *n* items and a sequence database *SDB*. The pattern *P* has a *maximal-backward-extension* in *SDB* if and only if for an integer *k* ($1 \leq k \leq n$), there exists an item *i* having a support higher or equal to *minsup* in the *k-th* maximum periods of *P* in SDB_P.

Definition 9. Let be a frequent sequential pattern *P* containing *n* items and a sequence database SDB_P. The pattern *P* has a *maximal-forward-extension* in *SDB* if and only if there exist an item *i* having a support higher or equal to *minsup* in the projected database SDB_P.

We now demonstrate that the maximal-forward-extension check and maximal-back-extension-check is sufficient for determining if a pattern is maximal.

Property 3. A frequent sequential pattern *P* is a maximal sequential pattern for a sequence database *SDB* if and only if it has no maximal-forward-extension in SDB_P and no maximal-backward-extension in the projected database SDB_P. **Rationale**. By definition, a pattern is maximal if it has no proper supersequence that is frequent. Consider a sequential pattern $P=\langle\{a_1\},\{a_2\}, \ldots,\{a_n\}\rangle$ containing *n* items. If there exists a proper supersequence of *P* that is frequent, it means that an item *i* can be added to *P* so that the resulting pattern *P'* would be frequent. It can be easily seen that the support of *P'* is the minimum of the support of *i* and the support of *P*. To count the support of *P'*, we can consider SDB_P instead of *SDB*, since *P'* can only appear in a subset of the sequences where *P* appear. To detect if such an item *i* can be appended to *P*, we need to consider three cases: (1) an item *i* can be appended to *P* before a_1, (2) an item *i* can be appended to *P* between any a_x and a_y such that $y = x+1$ and $1 \leq x \leq n$, or (3) an item *i* can be appended to *P* after a_n. The first two cases are verified by the *maximum backward extension check*. Verifying if an item can be appended before a_1 consists of verifying if an item is frequent in the first maximum period of *P* in SDB_P. Verifying if an item can be appended between any a_x and a_y such that $y = x+1$ and $1 \leq x \leq n$ is done by verifying if an item appear frequently in the *y*-th maximum period of *P* in SDB_P. Finally, the third case can be verified by simply counting the support of items in SDB_P to see if an item is frequent (because SDB_P is projected by *P* and thus items in SDB_P are items that can be appended to *P* after a_n).

Having shown that maximal-forward-extension and maximal-back-extension can be used to determine if a pattern is maximal, we next explain how they can be incorporated in the search procedure. We show the modified procedure in Figure 5, which we name MaxSP. It takes the same parameters as PrefixSpan: a current database *SDB*, a threshold *minsup* and a current prefix *P*. The algorithm first initializes a variable *largestSupport* to 0, which will be used to store the support of the largest pattern that can be obtained by extending the current prefix (line 1). Then, the algorithm counts the support of each item in the current database (line 2). Then for each frequent item *i*, the algorithm appends *i* at the end of prefix *P*, to obtain a new prefix *P'* and the projected database SDB_i is created by projecting SDB_P by *i* (line 3-5). Next, the check for maximal-back-extension is performed (line 6). If there is no maximal-back-extension, the procedure MaxSP is recursively called to explore patterns starting by *P'* (line 7). This recursive call returns the largest support of patterns that can be created by appending items to *P'*. If the largest support is smaller than *minsup*, it means that there is no maximal-forward-extension (line 8). In this case, the pattern *P'* is output (line 9). Next, the largest support for the prefix *P* is updated (line 10). Finally, after all frequent items have been processed, the largest support of extensions of *P* is returned.

3.3 Optimizations

We performed three optimizations to improve the performance of MaxSP. First, we use pseudo-projections instead of projections to avoid the cost of making physical database copies (as suggested in PrefixSpan [5]). A second optimization is to remove infrequent items from the database immediately after the first database scan because they will not appear in any maximal sequential pattern. A third optimization concerns the process of searching for the maximal-backward-extensions of a prefix by scanning maximum periods. During the scan, item supports are accumulated. The scan can be stopped as soon as there is an item known to appear in *minsup* maximum periods, because it means that the prefix is not maximal. For large databases containing many sequences, we found that this optimization increase performance by about a factor of two. Note that, this optimization has similarity to the ScanSkip optimization proposed in the BIDE algorithm, that stop scanning sequences as soon as a pattern is determined to be non closed [7].

```
MaxSP (a sequence database SDB, a threshold minsup, a prefix P initially set to ⟨⟩)
1.  largestSupport := 0.
2.  Scan SDB once to count the support of each item.
3.  FOR each item i with a support ≥ minsup
4.      P' := Concatenate(P, i).
5.      SDBi := DatabaseProjection(SDB, i ).
6.      IF the pattern P' has no maximal-backward-extension in SDBi THEN
7.          maximumSupport := MaxSP (SDB_i,  minsup, P').
8.          IF maximumSupport < minsup THEN OUTPUT the pattern  P'.
9.              IF support(P') > largestSupport THEN largestSupport := support(P')
10. RETURN largestSupport.
```

Fig. 5. The pseudocode of the MaxSP algorithm

4 Experimental Evaluation

To evaluate the performance of the proposed algorithm, we performed a set of experiments on a computer with a third generation Core i5 processor running Windows 7 and 1 GB of free RAM. We compared the performance of MaxSP with BIDE [7], a state-of-the-art algorithm for closed sequential pattern mining, which also does not store intermediate candidates during the mining process. We do not compare the performance of MaxSP with PrefixSpan because BIDE was previously shown to be more than an order of magnitude faster than PrefixSpan [6, 8]. All algorithms are implemented in Java and memory measurements were done by using the Java API. Five real-life datasets *BMS*, *Snake*, *Sign*, *Leviathan* and *FIFA* with varied characteristics are used in the experiments. Table 1 summarizes their characteristics and data types. The experiments consisted of running MaxSP and BIDE on each dataset while decreasing the minsup threshold until either an algorithm became too long to execute or a clear winner was found. For each dataset, we recorded the execution time, memory usage and the number of patterns found for each algorithm. Figure 6 and 7 respectively show the number of patterns found and the execution times for each dataset. Figure 8 presents the maximum memory usage for *Sign*, *Snake* and *Leviathan*. The source code of compared algorithms and datasets can be downloaded from http://goo.gl/hDtdt.

Table 1. Datasets' Characteristics

dataset	sequence count	distinct item count	avg. seq. length (items)	type of data
BMS	59,601	497	2.51 (std = 4.85)	web click stream
Snake	163	20	60 (std = 0.59)	protein sequences
Sign	730	267	51.99 (std = 12.3)	language utterances
Leviathan	5,834	9,025	33.81 (std= 18.6)	book
FIFA	20,450	2,990	34.74 (std = 24.08)	web click stream

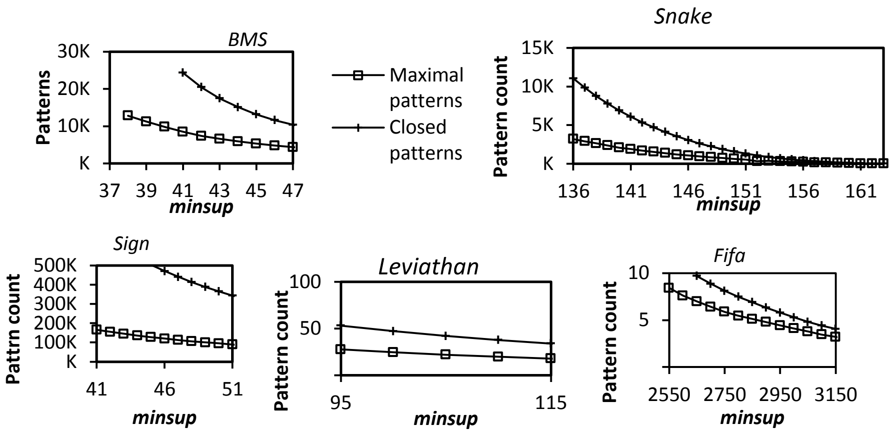

Fig. 6. Maximal pattern and closed pattern count

As it can be seen from the results, the number of maximal patterns is always considerably smaller than the number of closed patterns, and the gap increases quickly as *minsup* decreases. For example, for the *Sign* dataset, only 25 % of the closed sequential patterns are maximal for *minsup* = 47. Another example is *Snake*, where only 28 % of the closed patterns are maximal for *minsup* = 136. This confirms that mining maximal sequential patterns is more efficient in terms of storage space.

With respect to execution time, we can see that MaxSP is always faster than BIDE. The difference is large for sparse datasets such as *Sign* and *FIFA*. For example, for *Sign*, MaxSP was up to five times faster than BIDE. There are two reasons why MaxSP is faster. The first reason is how the maximal-backward-extension checking is performed. For each pattern found, MaxSP looks for items that could extend it with a support no less than minsup, while BIDE looks for items with a support equal to the support of the prefix. As soon as MaxSP or BIDE find an item meeting their respective conditions, they stop searching for backward extensions (the third optimization in MaxSP and the ScanSkip optimization in BIDE). Because the condition verified by BIDE is more specific and thus harder to meet, BIDE needs to analyze more sequences on average for backward extension checking, and this makes BIDE slower. The second reason is that BIDE needs to perform more write operations to disk for storing patterns because the set of closed sequential patterns is larger.

For the memory usage (cf. Figure 8), similar conclusions can be drawn. MaxSP generally uses less memory than BIDE. This is due to the fact that less sequences need to be scanned (as explained previously) and less patterns need to be created by MaxSP. Note that we did not show the memory usage for FIFA and BMS due to space limitation but results are similar as those of Leviathan, Snake and Sign.

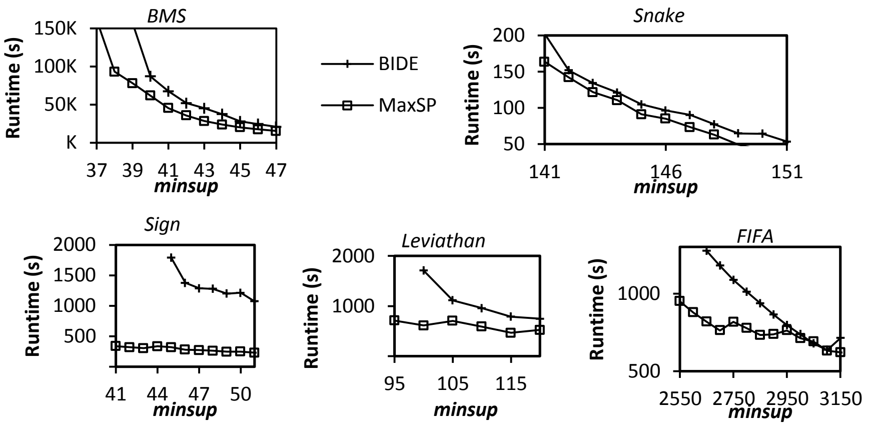

Fig. 7. Execution times of MaxSP and BIDE

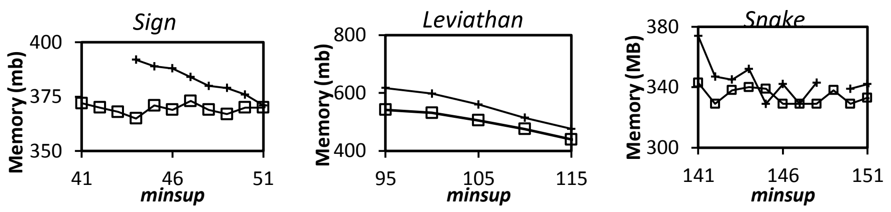

Fig. 8. Maximum memory usage for Sign, Leviathan and Snake

5 Conclusion

Maximal sequential pattern mining is an important data mining task that is essential for a wide range of applications. The set of maximal sequential patterns is a compact representation of sequential patterns. Several algorithms have been proposed for maximal sequential pattern mining. However, they are not memory efficient since they may produce too many redundant candidates and need to maintain a large amount of intermediate candidates in main memory during the mining process. To address these problems, we proposed a memory efficient algorithm to mine maximal sequential patterns named MaxSP (*M*aximal *S*equential *P*attern miner). It incorporates a novel checking mechanism consisting of verifying maximal-backward-extensions and maximal-forward-extensions, which allows discovering all maximal sequential patterns without storing intermediate candidates in memory nor producing redundant candidates. An experimental study on five real datasets shows that MaxSP is more memory efficient and up to five time faster than BIDE, a state-of-art algorithm for closed sequential pattern mining, and that the number of maximal patterns is generally much smaller than the number of closed sequential patterns. The source code of MaxSP and BIDE be downloaded from http://goo.gl/hDtdt as part of the SPMF open-source data mining software.

For future work, we plan to develop new algorithms for mining concise representations of sequential patterns and sequential rules [15, 16].

References

1. Han, J. and Kamber, M.: Data Mining: Concepts and Techniques, 2nd ed., San Francisco, Morgan Kaufmann (2006)
2. Agrawal, R. and Srikant, R.: Mining Sequential Patterns, Proc. Int. Conf. on Data Engineering, pp. 3-14 (1995)
3. 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)
4. Ayres, J., Flannick, J., Gehrke, J. and Yiu, T.: Sequential PAttern mining using a bitmap representation, Proc. KDD 2002, Edmonton, Alberta, pp. 429-435 (2002)
5. 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)
6. Zaki, M. J.: SPADE: An efficient algorithm for mining frequent sequences, Machine learning, vol. 42. no. 1-2, pp. 31-60 (2001)
7. Wang, J., Han, J., Li, C.: Frequent Closed Sequence Mining without Candidate Maintenance, IEEE Trans. on Know. and Data Engineering, vol. 19, no. 8, pp.1042-1056 (2007)
8. Yan, X., Han, J. and Afshar, R.: CloSpan: Mining closed sequential patterns in large datasets, Proc. of the third SIAM International Conference on Data Mining, May 1-3, San Francisco, California, ISBN 0-89871-545-8. (2003)
9. Gomariz, A., Campos, M., Marin, R., Goethals, B.: ClaSP: An Efficient Algorithm for Mining Frequent Closed Sequences, Proc. PAKDD 2013, LNAI 7818, pp. 50-61 (2013)
10. García-Hernández, R. A., Martínez-Trinidad, J. F., Carrasco-Ochoa, J. A.: A new algorithm for fast discovery of maximal sequential patterns in a document collection. Comp. Linguistics and Intelligent Text Processing, Springer LNCS 3878, pp. 514-523 (2006)
11. Lin, N. P., Hao, W.-H., Chen, H.-J., Chueh, H.-E., Chang, C.-I.: Fast Mining Maximal Sequential Patterns. Proc. of the 7th International Conference on Simulation, Modeling and Optimization, September 15-17, Beijing, China, pp.405-408 (2007)
12. Luo, C., Chung, S.: Efficient mining of maximal sequential patterns using multiple samples." Proc. 5th SIAM int'l conf. on data mining, Newport Beach, California. (2005)
13. Lu, S., Li, C.: AprioriAdjust: An Efficient Algorithm for Discovering the Maximum Sequential Patterns, Proc. 2nd Int'l Workshop Knowl. Grid and Grid Intell. (2004)
14. Guan, E.-Z., Chang, X.-Y., Wang, Z., Zhou, C.-G.: Mining Maximal Sequential Patterns, Proc of the second Int'l Conf. Neural Networks and Brain, pp.525-528 (2005)
15. Fournier-Viger, P., Nkambou, R., Tseng, V. S.: RuleGrowth: Mining Sequential Rules Common to Several Sequences by Pattern-Growth. Proc. of the 26th Symposium on Applied Computing. Tainan, Taiwan, pp. 954-959, ACM Press (2011).
16. Fournier-Viger, P., Faghihi, U., Nkambou, R., Mephu Nguifo, E.: CMRules: Mining Sequential Rules Common to Several Sequences. Knowledge-based Systems, Elsevier, vol. 25, no. 1, pp. 63-76 (2012)