VGEN: Fast Vertical Mining of Sequential Generator Patterns

Philippe Fournier-Viger[1], Antonio Gomariz[2], Michal Šebek[3], Martin Hlosta[3]

[1] Dept. of Computer Science, University of Moncton, Canada
[2] Dept. of Information and Communication Engineering, University of Murcia, Spain
[3] Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic

`philippe.fournier-viger@umoncton.ca`, `agomariz@um.es`, `{isebek,ihlosta}@fit.vutbr.cz`

Abstract. *Sequential pattern mining* is a popular data mining task with wide applications. However, the set of all sequential patterns can be very large. To discover fewer but more representative patterns, several compact representations of sequential patterns have been studied. The set of *sequential generator patterns* is one the most popular representations. However, although mining sequential generator patterns is key to several other data mining tasks such as sequential rule mining, it remains very computationally expensive. To address this issue, we introduce a novel mining algorithm named *VGEN* (*Vertical Sequential GENerator Pattern Miner*). An experimental study on five real datasets shows that VGEN outperforms several state-of-the-art sequential pattern mining algorithms.

Keywords: sequential patterns, generators, vertical mining, candidate pruning

1 Introduction

Mining interesting patterns in sequential data is a challenging task. Multiple studies have been proposed for mining interesting patterns in sequence databases [11, 5]. *Sequential pattern mining* is probably the most popular research topic among them. A subsequence 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* [1]. 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 [11].

Several algorithms have been proposed for sequential pattern mining such as *PrefixSpan* [12], *SPAM* [3] and *SPADE* [18]. However, a critical 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 the user. This will most likely make the algorithm fail to terminate in reasonable time and run out of memory. For example, the well-known PrefixSpan [12] algorithm 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 more representative patterns to users, many studies focus on developing concise representations of sequential patterns. A popular representation that has been proposed is *closed sequential patterns* [15, 16, 6]. A closed sequential pattern is a sequential pattern that is not strictly included in another pattern having the same frequency. Another important representation is *sequential generator patterns*. A sequential generator is a sequential pattern such that there does not exist a smaller pattern with the same support that is included in it. Thus, generator and closed patterns are related. They are respectively the minimal and maximal elements of equivalence classes of patterns having the same support, partially ordered by the inclusion relation.

Mining sequential generator patterns is desirable for several reasons. First, it was proven that the set of sequential generators is often smaller than the set of closed sequential patterns. The reason is that each equivalence class may have multiple generators and closed patterns[4] [9]. Second, sequential generators are more suitable according to the principles of the MDL (Minimum Description Length) as they represent the minimal elements of equivalence classes rather the maximal ones [9]. Third, mining sequential generators is key to other important data mining tasks such as sequential rule mining, where rules having a minimum antecedent and maximum consequent can be formed using sequential generators and closed patterns [10].

Although mining sequential generators is desirable and useful in many applications, it remains a computationally expensive data mining task and few algorithms have been proposed for this task. Most algorithms such as GenMiner [9], FEAT [8] and FSGP [17] employs a pattern-growth algorithms by extending the PrefixSpan algorithm [12]. These algorithms only differ by how patterns are stored and whether generators are identified on-the-fly or by post-processing. Because these algorithms all adopts a pattern-growth approach, they have comparable performance to other pattern-growth algorithms. For example, GenMiner was shown to have approximately the same performance as other pattern-growth algorithms CloSpan [16] and BIDE [15], and FEAT was shown to outperforms PrefixSpan by about an order of magnitude, which is thus also comparable to CloSpan and BIDE. Because these algorithms use a pattern-growth approach, they also suffer from its main limitation which is to repeatedly perform database

[4] This is different from frequent itemset mining where each equivalence class has only one closed pattern.

projections to grow patterns, which is an extremely costly operation (in the worst case, a pattern-growth algorithm will perform a database projection for each item of each frequent pattern). More recently, the MGSP algorithm and has similar performance to FSGP [13].

Given the limitation of these previous works, we present a novel algorithm for mining sequential generator patterns that we name *VGEN* (*Vertical Sequential GENerator Miner*). VGEN rely on a depth-first exploration of the search space using a vertical representation of the database. The algorithm incorporates three efficient strategies named ENG (Efficient filtering of Non-Generator patterns), BEC (Backward Extension checking) and CPC (Candidate Pruning by Co-occurrence map) to effectively identify generator patterns and prune the search space.

VGEN can capture the complete set of sequential generators by performing a single database scan (to build the vertical structure). We performed an experimental study with five real-life datasets to compare the performance of VGEN with state-of-the-art sequential pattern mining algorithms. Results show that VGEN outperforms those algorithms by up to two orders of magnitude.

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

2 Problem Definition

Definition 1 (sequence database). Let $I = \{i_1, i_2, ..., i_l\}$ be a set of items (symbols). An *itemset* $I_x = \{i_1, i_2, ..., i_m\} \subseteq I$ is an unordered set of distinct items. The *lexicographical order* $\succ_{lex}$ is defined as any total order on I. Without loss of generality, we assume that all itemsets are ordered according to $\succ_{lex}$. A *sequence* is an ordered list of itemsets $s = \langle I_1, I_2, ..., I_n \rangle$ such that $I_k \subseteq I$ $(1 \leq k \leq n)$. A *sequence database SDB* is a list of sequences $SDB = \langle s_1, s_2, ..., s_p \rangle$ having sequence identifiers (SIDs) $1, 2...p$. **Example.** A sequence database is shown in Fig. 1 (left). It contains four sequences having the SIDs 1, 2, 3 and 4. Each single letter represents an item. Items between curly brackets represent an itemset. The first sequence $\langle \{a, b\}, \{c\}, \{f, g\}, \{g\}, \{e\} \rangle$ contains five itemsets. It indicates that items a and b occurred at the same time, were followed by c, then f and g at the same time, followed by g and lastly e.

Definition 2 (sequence containment). A sequence $s_a = \langle A_1, A_2, ..., A_n \rangle$ is said to be *contained in* a sequence $s_b = \langle B_1, B_2, ..., B_m \rangle$ iff there exist 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 $s_a \sqsubseteq s_b$). **Example.** Sequence 4 in Fig. 1 (left) is contained in Sequence 1.

Definition 3 (prefix). A sequence $s_a = \langle A_1, A_2, ..., A_n \rangle$ is a *prefix* of a sequence $s_b = \langle B_1, B_2, ..., B_m \rangle$, $\forall n < m$, iff $A_1 = B_1, A_2 = B_2, ..., A_{n-1} = B_{n-1}$ and the first $|A_n|$ items of B_n according to $\succ_{lex}$ are equal to A_n.

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

Pattern	Sup.	Pattern	Sup.
$\langle\{a\}\rangle$	3 CG	$\langle\{b\},\{g\},\{e\}\rangle$	2 C
$\langle\{a\},\{g\}\rangle$	2 G	$\langle\{b\},\{f\}\rangle$	4 C
$\langle\{a\},\{g\},\{e\}\rangle$	2 C	$\langle\{b\},\{f, g\}\rangle$	2 C
$\langle\{a\},\{f\}\rangle$	3 C	$\langle\{b\},\{f\},\{e\}\rangle$	2 C
$\langle\{a\},\{f\},\{e\}\rangle$	2 C	$\langle\{b\},\{e\}\rangle$	3 C
$\langle\{a\},\{c\}\rangle$	2	$\langle\{c\}\rangle$	2 G
$\langle\{a\},\{c\},\{f\}\rangle$	2 C	$\langle\{c\},\{f\}\rangle$	2
$\langle\{a\},\{c\},\{e\}\rangle$	2 C	$\langle\{c\},\{e\}\rangle$	2
$\langle\{a\},\{b\}\rangle$	2 G	$\langle\{e\}\rangle$	3 G
$\langle\{a\},\{b\},\{f\}\rangle$	2 C	$\langle\{f\}\rangle$	4
$\langle\{a\},\{b\},\{e\}\rangle$	2 C	$\langle\{f, g\}\rangle$	2 G
$\langle\{a\},\{e\}\rangle$	3 C	$\langle\{f\},\{e\}\rangle$	2 G
$\langle\{a, b\}\rangle$	2 CG	$\langle\{g\}\rangle$	3 G
$\langle\{b\}\rangle$	4	$\langle\{g\},\{e\}\rangle$	2 G
$\langle\{b\},\{g\}\rangle$	3 C	$\langle\rangle$	4 G

C = Closed G = Generator

Fig. 1. A sequence database (left) and (all/closed/generator) sequential patterns found (right)

Definition 4 (support). The *support* of a sequence s_a in a sequence database SDB is defined as the number of sequences $s \in SDB$ such that $s_a \sqsubseteq s$ and is denoted by $sup_{SDB}(s_a)$.

Definition 5 (sequential pattern mining). Let *minsup* be a threshold set by the user and SDB be a sequence database. A sequence s is a *sequential pattern* and is deemed *frequent* iff $sup_{SDB}(s) \geq minsup$. The *problem of mining sequential patterns* is to discover all sequential patterns [1]. **Example.** Fig. 1 (right) shows the 30 sequential patterns found in the database of Fig. 1 (left) for $minsup = 2$, and their support. For instance, the patterns $\langle\{a\}\rangle$, $\langle\{a\}, \{g\}\rangle$ and $\langle\rangle$ (the empty sequence) are frequent and have respectively a support of 3, 2 and 4 sequences.

Definition 6 (closed/generator sequential pattern mining). A sequential pattern s_a is said to be *closed* if there is no other sequential pattern s_b, such that s_b is a superpattern of s_a, $s_a \sqsubseteq s_b$, and their supports are equal. A sequential pattern s_a is said to be a *generator* if there is no other sequential pattern s_b, such that s_a is a superpattern of s_b, $s_b \sqsubseteq s_a$, and their supports are equal. The *problem of mining closed (generator) sequential patterns* is to discover the set of closed (generator) sequential patterns. **Example.** Consider the database of Fig. 1 and minsup = 2. There are 30 sequential patterns (shown in the right side of Fig. 1), such that 15 are closed (identified by the letter C) and only 11 are generators (identified by the letter G). It can be observed that in this case, the number of generators is less than the number of closed patterns.

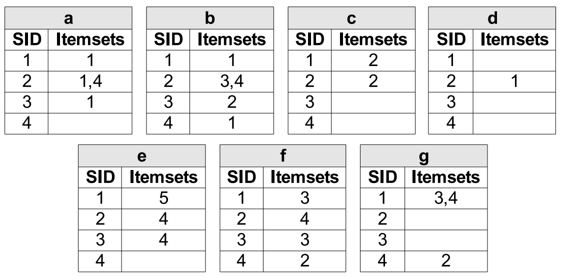
We next present an important pruning property for sequential pattern generators, which is used by previous algorithms for sequential generator mining [9].

Definition 7 (database projection). The projection of a sequence database SDB by a sequence s_a is denoted as SDB_{s_a} and defined as the projection of all sequences from SDB by s_a. Let be two sequences $s_a = \langle A_1, A_2, ..., A_n \rangle$ and $s_b = \langle B_1, B_2, ..., B_m \rangle$. If $s_a \sqsubset s_b$, the projection of s_b by s_a is defined as $\langle B_{k1}, B_{k2} ... B_{km} \rangle$ for the smallest integers $0 < k1 < k2 < ...km \leq m$ such that $A_1 \subseteq B_{k1}$, $A_2 \subseteq B_{k2}$, ... $A_2 \subseteq B_{km}$. Otherwise, the projection is undefined.

Property 1. (non-generator pruning property). Given a sequence database SDB, let s_a and s_b be two sequential patterns. If $s_b \sqsubset s_a$ and the projection of the database SDB by s_a and s_b results in the same sequence database, then s_a and any extensions of s_a at not generators [9].

Definition 8 (horizontal database format). A *sequence database in horizontal format* is a database where each entry is a sequence. **Example.** Figure 1 (left) shows an horizontal sequence database.

Definition 9 (vertical database format). A *sequence database in vertical format* is a database where each entry represents an item and indicates the list of sequences where the item appears and the position(s) where it appears [3]. **Example.** Fig. 2 shows the vertical representation of the database of Fig. 1 (left).



a	
SID	Itemsets
1	1
2	1,4
3	1
4	

b	
SID	Itemsets
1	1
2	3,4
3	2
4	1

c	
SID	Itemsets
1	2
2	2
3	
4	

d	
SID	Itemsets
1	
2	1
3	
4	

e	
SID	Itemsets
1	5
2	4
3	4
4	

f	
SID	Itemsets
1	3
2	4
3	3
4	2

g	
SID	Itemsets
1	3,4
2	
3	
4	2

Fig. 2. The vertical representation of the database shown in Fig. 1(left).

Vertical mining algorithms associate a structure named *IdList* [18, 3] to each pattern. IdLists allow calculating the support of a pattern quickly by making join operations with IdLists of smaller patterns. To discover sequential patterns, vertical mining algorithms perform a single database scan to create IdLists of patterns containing single items. Then, larger patterns are obtained by performing the join operation of IdLists of smaller patterns (cf. [18] for details). Several works proposed alternative representations for IdLists to save time in join operations, being the bitset representation the most efficient one [3].

3 The VGEN Algorithm

We present VGEN, our novel algorithm for sequential generator mining. It adopts the IdList structure [3, 6, 18]. We first describe the general search procedure used by VGEN to explore the search space of sequential patterns. Then, we describe how it is adapted to discover sequential generators efficiently.

3.1 The search procedure

The pseudocode of the search procedure is shown in Fig. 3. The procedure takes as input a sequence database SDB and the *minsup* threshold. The procedure first scans the input database SDB once to construct the vertical representation of the database $V(SDB)$ and the set of frequent items F_1. For each frequent item $s \in F_1$, the procedure calls the SEARCH procedure with $\langle s \rangle$, F_1, $\{e \in F_1 | e \succ_{lex} s\}$, and *minsup*.

The SEARCH procedure outputs the pattern $\langle \{s\} \rangle$ and recursively explores candidate patterns starting with the prefix $\langle \{s\} \rangle$. The SEARCH procedure 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 second set S_i represents items to be appended to *pat* by *i*-extension. For each candidate pat' generated by an extension, the procedure calculate the support to determine if it is frequent. This is done by the IdList join operation (see [3, 18] for details) and counting the number of sequences where the pattern appears. If the pattern pat' is frequent, it is then used in a recursive call to SEARCH to generate patterns starting with the prefix pat'.

It can be easily seen that the above procedure is correct and complete to explore the search space of sequential patterns since it starts with frequent patterns containing single items and then extend them one item at a time while only pruning infrequent extensions of patterns using the anti-monotonicity property (any infrequent sequential pattern cannot be extended to form a frequent pattern)[1].

3.2 Discovering sequential generator patterns

We now describe how the search procedure is adapted to discover only generator patterns. This is done by integrating three strategies to efficiently filter non-generator patterns and prune the search space. The result is the VGEN algorithm, which outputs the set of generator patterns.

Strategy 1. Efficient filtering of Non-Generator patterns (ENG). The first strategy identifies generator patterns among patterns generated by the search procedure. This is performed using a novel structure named Z that stores the set of generator patterns found until now. The structure Z is initialized as a set containing the empty sequence $\langle\rangle$ with its support equal to $|SDB|$. Then, during the search for patterns, every time that a pattern s_a, is generated by the search procedure, two operations are performed to update Z.

```
PATTERN-ENUMERATION(SDB, minsup)
1.  Scan SDB to create V(SDB) and identify $S_{init}$, the list of frequent items.
2.  FOR each item s ∈ $S_{init,}$
3.      SEARCH(⟨s⟩, $S_{init}$, the set of items from $S_{init}$ that are lexically larger than s, minsup).
```

```
SEARCH(pat, $S_{n,}$ $I_n$, minsup)
1.  Output pattern pat.
2.  $S_{temp}$ := $I_{temp}$ := ∅
3.  FOR each item j ∈ $S_{n,}$
4.      IF the s-extension of pat is frequent THEN $S_{temp}$ := $S_{temp}$ ∪{i}.
5.  FOR each item j∈ $S_{temp,}$
6.      SEARCH(the s-extension of pat with j, $S_{temp\ ,}$ elements in $S_{temp}$ greater than j, min-
sup).
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.     SEARCH(i-extension of pat with j, $S_{temp\ ,}$ all elements in $I_{temp}$ greater than j, minsup).
```

Fig. 3. The search procedure

- *Sub-pattern checking.* During this operation, s_a is compared with each pattern $s_b \in Z$ to determine if there exists a pattern s_b such that $s_b \sqsubset s_a$ and $sup(s_a) = sup(s_b)$. If yes, then s_a is not a generator (by Definition 6) and thus, s_a is not inserted into Z. Otherwise, s_a is a generator with respect to all patterns found until now and it is thus inserted into Z.
- *Super-pattern checking.* If s_a is determined to be a generator according to super-pattern checking, we need to perform this second operation. The pattern s_a is compared with each pattern $s_b \in Z$. If there exists a pattern s_b such that $s_a \sqsubseteq s_b$ and $sup(s_a) = sup(s_b)$, then s_b is not a generator (by Definition 6) and s_b is removed from Z.

By using the above strategy, it is obvious that when the search procedure terminates, Z contains the set of sequential generator patterns. However, to make this strategy efficient, we need to reduce the number of pattern comparisons and containment checks ($\sqsubseteq$). We propose five optimizations.

1. *Size check optimization.* Let n be the number of items in the largest pattern found until now. The structure Z is implemented as a list of maps $Z = \{M_1, M_2, ...M_n\}$, where M_x contains all generator patterns found until now having x items ($1 \leq x \leq n$). To perform sub-pattern checking (super-pattern checking) for a pattern s containing w items, an optimization is to only compare s with patterns in $M_1, M_2...M_{w-1}$ (in $M_{w+1}, M_{w+2}...M_n$) because a pattern can only contain (be contained) in smaller (larger) patterns.
2. *SID count optimization.* To verify the pruning property 1, it is required to compare pairs of patterns s_a and s_b to see if their projected databases are identical, which will be presented in the BEC strategy. A necessary condition to have identical projected databases is that the sum of SIDs (Sequence IDs) containing s_a and s_b is the same. To check this condition efficiently, the sum

of SIDs is computed for each pattern and each map M_k contains mappings of the form $(l, S_k$ where S_k is the set of all patterns in Z having l as sum of SIDS (Sequence IDs).

3. *Sum of items optimization.* In our implementation, each item is represented by an integer. For each pattern s, the *sum of the items* appearing in the pattern is computed, denoted as $sum(s)$. This allows the following optimization. Consider super-pattern checking for pattern s_a and s_b. If $sum(s_a) < sum(s_b)$ for a pattern s_b, then we don't need to check $s_a \sqsubseteq s_b$. A similar optimization is done for sub-pattern checking. Consider sub-pattern checking for a pattern s_a and a pattern s_b. If $sum(s_b) < sum(s_a)$ for a pattern s_b, then we don't need to check $s_b \sqsubseteq s_a$.
4. *Support check optimization.* This optimization uses the support to avoid containment checks ($\sqsubseteq$). If the support of a pattern s_a is less than the support of another pattern s_b (greater), then we skip checking $s_a \sqsubseteq s_b$ ($s_b \sqsubseteq s_a$).
5. Lastly, another optimizations is to compare the sum of even and odd items instead of the sum of items.

Strategy 2. Backward Extension checking (BEC). The second strategy aims at avoiding sub-pattern checks. The search procedure discovers patterns by growing a pattern by appending one item at a time by s-extension or i-extension. Consider a pattern x' that is generated by extension of a pattern x. An optimization is to not perform sub-pattern checking if x' has the same support as x (because this pattern would have x has prefix, thus indicating that x is not a generator).

Furthermore, another related optimization is to implement the pruning property 1. For a pattern x, this is done during sub-pattern checking. If a smaller pattern y can be found such that the projected database is identical, then any extension of x should not be explored. Checking if projected databases are identical is simply performed by comparing the IdLists of x and y.

Strategy 3. Candidate Pruning with Co-occurrence map (CPC). The last strategy aims at pruning the search space of patterns by exploiting item co-occurrence information. We introduce a structure named *Co-occurrence MAP* (CMAP) defined as follows: an item k is said to *succeed by i-extension* to an item j in a sequence $\langle I_1, I_2, ..., I_n \rangle$ iff $j, k \in I_x$ for an integer x such that $1 \leq x \leq n$ and $k \succ_{lex} j$. In the same way, an item k is said to *succeed by s-extension* to an item j in a sequence $\langle I_1, I_2, ..., I_n \rangle$ iff $j \in I_v$ and $k \in I_w$ for some integers v and w such that $1 \leq v < w \leq n$. A CMAP is a structure mapping each item $k \in I$ to a set of items succeeding it.

We define two CMAPs named $CMAP_i$ and $CMAP_s$. $CMAP_i$ maps each item k to the set $cm_i(k)$ of all items $j \in I$ succeeding k by i-extension in no less than *minsup* sequences of SDB. $CMAP_s$ maps each item k to the set $cm_s(k)$ of all items $j \in I$ succeedings k by s-extension in no less than *minsup* sequences of SDB. For example, the $CMAP_i$ and $CMAP_s$ structures built for the sequence database of Fig. 1(left) are shown in Table 1. Both tables have been created considering a *minsup* of two sequences. For instance, for the item f, we can see that it is associated with an item, $cm_i(f) = \{g\}$, in $CMAP_i$, whereas it is associated with two items, $cm_s(f) = \{e, g\}$, in $CMAP_s$. This indicates that both items e and g succeed to f by s-extension and only item g does the same for i-extension, being all of them in no less than *minsup* sequences.

VGEN uses CMAPs to prune the search space as follows:

1. *s-extension(s) pruning.* Let a sequential pattern *pat* being considered for *s*-extension with an item $x \in S_n$ by the SEARCH procedure (line 3). If the last item a in *pat* does not have an item $x \in cm_s(a)$, then clearly the pattern resulting from the extension of *pat* with x will be infrequent and thus the join operation of x with *pat* to count the support of the resulting pattern does not need to be performed. Furthermore, the item x is not considered for generating any pattern by s-extension having *pat* as prefix, by not adding x to the variable S_{temp} that is passed to the recursive call to the SEARCH procedure. Moreover, note that we only have to check the extension of *pat* with x for the last item in *pat*, since other items have already been checked for extension in previous steps.
2. *i-extension(s) pruning.* Let a sequential pattern *pat* being considered for *i*-extension with an item $x \in I_n$ by the SEARCH procedure. If the last item a in *pat* does not have an item $x \in cm_i$, then clearly the pattern resulting from the extension of *pat* with x will be infrequent and thus the join operation of x with *pat* to count the support of the resulting pattern does not need to be performed. Furthermore, the item x is not considered for generating any pattern by i-extension(s) of *pat* by not adding x to the variable I_{temp} that is passed to the recursive call to the SEARCH procedure. As before, we only have to check the extension of *pat* with x for the last item in *pat*, since others have already been checked for extension in previous steps.

CMAPs are easily maintained and are built with a single database scan. With regards to their implementation, we define each one as a hash table of hash sets, where an hashset corresponding to an item k only contains the items that succeed to k in at least *minsup* sequences.

$CMAP_i$		$CMAP_s$	
item	is succeeded by (i-extension)	item	is succeeded by (s-extension)
a	$\{b\}$	a	$\{b, c, e, f\}$
b	$\emptyset$	b	$\{e, f, g\}$
c	$\emptyset$	c	$\{e, f\}$
e	$\emptyset$	e	$\emptyset$
f	$\{g\}$	f	$\{e, g\}$
g	$\emptyset$	g	$\emptyset$

Table 1. $CMAP_i$ and $CMAP_s$ for the database of Fig. 1 and $minsup = 2$.

Lastly, since the VGEN algorithm is a vertical mining algorithm, it relies on IDLists. We implement IDLists as bitsets as it is done in several state-of-the-art algorithms [3, 6]. Bitsets speed up the join operations. Algorithms using this representation were demonstrated to be much faster than vertical mining algorithms which do not use them.

4 Experimental Evaluation

We performed several experiments to assess the performance of the proposed algorithm. Experiments were performed on a computer with a third generation Core i5 64 bit processor running Windows 7 and 5 GB of free RAM. We compared the performance of VGEN with .... current state-of-the-art algorithm for sequential pattern mining. All algorithms were implemented in Java. All memory measurements were done using the Java API. Experiments were carried on five real-life datasets having varied characteristics and representing three different types of data (web click stream, text from a book and protein sequences). Those datasets are *Leviathan*, *Snake*, *FIFA*, *BMS* and *Kosarak10k*. Table 2 summarizes their characteristics. The source code of all algorithms and datasets used in our experiments can be downloaded from `http://goo.gl/R32D9d`.

dataset	sequence count	item count	avg. seq. length (items)	type of data
Leviathan	5834	9025	33.81 (std= 18.6)	book
Snake	163	20	60 (std = 0.59)	protein sequences
FIFA	20450	2990	34.74 (std = 24.08)	web click stream
BMS	59601	497	2.51 (std = 4.85)	web click stream
Kosarak10k	10000	10094	8.14 (std = 22)	web click stream

Table 2. Dataset characteristics

Experiment 1. Influence of the *minsup* parameter. The first experiment consisted of running all the algorithms on each dataset while decreasing the *minsup* threshold until an algorithm became too long to execute, ran out of memory or a clear winner was observed. For each dataset, we recorded the execution time and memory usage.

In terms of execution time, results (cf. Fig. 4) show that VGEN outperforms .... by a wide margin on all datasets.

In terms of memory consumption the maximum memory usage of VGEN.....

Experiment 2. Influence of the strategies. We next evaluated the benefit of using strategies in VGEN. We compared VGEN with a version of VGEN without strategy CPC (VGEN_WC) and a version without strategy BEC (VGEN_WB). Results for the Kosarak and Leviathan datasets are shown in Fig. 5. Results for other datasets are similar and are not shown due to space limitation. As a whole, strategies improved execution time by up to to 8 times, CPC being the most effective strategy.

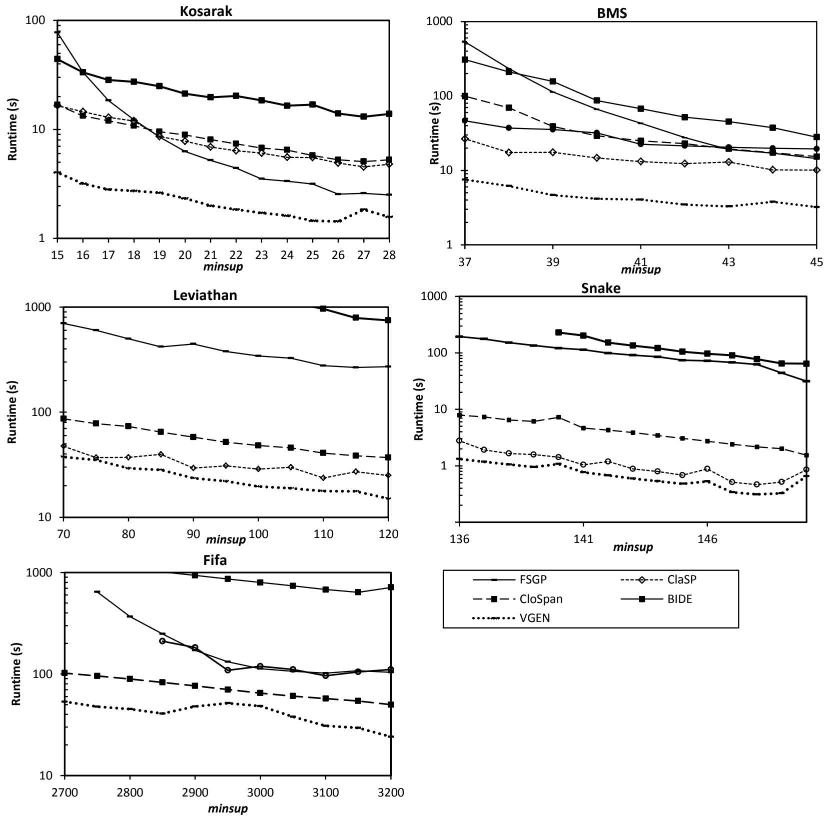


Fig. 4. Execution times

We also measured the memory used by the CPC strategy to build the VGEN data structure. We found that the required amount memory is very small. For the BMS, Kosarak, Leviathan, Snake and FIFA datasets, the memory footprint of CMAPs was respectively 0.5 MB, 33.1 MB, 15 MB, 64 KB and 0.4 MB.

5 Conclusion

In this paper, we presented a new sequential generator pattern mining algorithm named VGEN (Vertical Sequential GENerator Pattern Miner). It is to our knowledge the first vertical algorithm for this task. Furthermore, it includes three novel strategies for efficiently identifying generator patterns and pruning the search space (ENG, BEC and CPC). An experimental study on five real datasets shows that VGEN is up to two orders of magnitude faster than MaxSP, the state-of-art algorithm for maximal sequential pattern mining, and that VGEN performs well on dense datasets. The source code of VGEN and all compared algorithms can be downloaded from `http://goo.gl/R32D9d`.

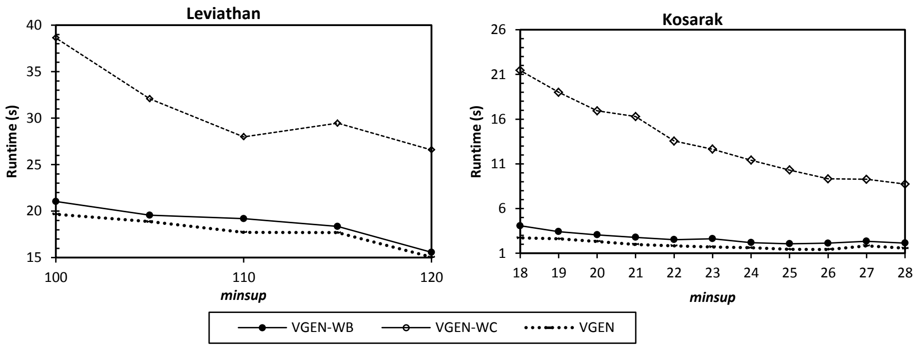


Fig. 5. Influence of optimizations for BMS (left) and FIFA (right)

References

1. Agrawal, R., Ramakrishnan, S.: Mining sequential patterns. In: Proc. 11th Intern. Conf. Data Engineering, pp. 3–14. IEEE (1995)
2. Aseervatham, S., Osmani, A., Viennet, E.: bitSPADE: A Lattice-based Sequential Pattern Mining Algorithm Using Bitmap Representation. In: Proc. 6th Intern. Conf. Data Mining, pp.792–797. IEEE (2006)
3. Ayres, J., Flannick, J., Gehrke, J., Yiu, T.: Sequential pattern mining using a bitmap representation. In: Proc. 8th ACM Intern. Conf. Knowl. Discov. Data Mining, pp. 429–435. ACM (2002)
4. Fournier-Viger, P., Wu, C.-W., Tseng, V.-S.: Mining Maximal Sequential Patterns without Candidate Maintenance. In: Proc. 9th Intern. Conference on Advanced Data Mining and Applications, Springer, LNAI 8346, pp. 169-180 (2013)
5. Fournier-Viger, P., Gomariz, A., Campos, M., Thomas, R.: Fast Vertical Sequential Pattern Mining Using Co-occurrence Information. In: Proc. 18th Pacific-Asia Conference on Knowledge Discovery and Data Mining , Springer, LNAI, (2014)
6. Gomariz, A., Campos, M., Marin, R., Goethals, B.: ClaSP: An Efficient Algorithm for Mining Frequent Closed Sequences. In: Proc. 17th Pacific-Asia Conf. Knowledge Discovery and Data Mining, pp. 50–61. Springer (2013)
7. Lin, N. P., Hao, W.-H., Chen, H.-J., Chueh, H.-E., Chang, C.-I.: Fast Mining Maximal Sequential Patterns. In: Proc. of the 7th Intern. Conf. on Simulation, Modeling and Optimization, September 15-17, Beijing, China, pp.405-408 (2007)
8. Gao, C., Wang, J., He, Y., Zhou, L.: Efficient mining of frequent sequence generators. In: Proc. 17th Intern. Conf. World Wide Web:, pp. 1051–1052 (2008)
9. Lo, D., Khoo, S.-C., Li, J.: Mining and Ranking Generators of Sequential Patterns. In: Proc. SIAM Intern. Conf. Data Mining 2008, pp. 553–564 (2008)
10. Lo, D., Khoo, S.-C., Wong, L.: Non-redundant sequential rulesTheory and algorithm. Information Systems 34(4), 438–453 (2009)
11. Mabroukeh, N.R., Ezeife, C.I.: A taxonomy of sequential pattern mining algorithms, ACM Computing Surveys 43(1), 1–41 (2010)
12. Pei, J., Han, J., Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Dayal, U., Hsu, M.: Mining sequential patterns by pattern-growth: the PrefixSpan approach. IEEE Trans. Known. Data Engin. 16(11), 1424–1440 (2004)

13. Pham, T.-T., Luo, J., Hong, T.-P., Vo, B..: MSGPs: a novel algorithm for mining sequential generator patterns. In: Proc. 4th Intern. Conf. Computational Collective Intelligence, pp. 393-401 (2012)
14. Szathmary, L., Valtchev, P., Napoli, A., Godin., R.: Efficient vertical mining of frequent closures and generators. In: Proc. 8th Intern. Symp. Intelligent Data Analysis, August 31 - September 2, Lyon, France, pp. 393–404 (2009)
15. Wang, J., Han, J., Li, C.: Frequent closed sequence mining without candidate maintenance. IEEE Trans. on Knowledge Data Engineering 19(8), 1042–1056 (2007)
16. Yan, X., Han, J., Afshar, R.: CloSpan: Mining closed sequential patterns in large datasets. In: Proc. 3rd SIAM Intern. Conf. on Data Mining, pp. 166–177 (2003)
17. Yi, S., Zhao, T., Zhang, Y., Ma, S., Che, Z.: An effective algorithm for mining sequential generators. Procedia Engineering, 15, 3653-3657 (2011)
18. Zaki, M.J.: SPADE: An efficient algorithm for mining frequent sequences. Machine Learning 42(1), 31–60 (2001)