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, 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 generators* is one the most popular representations. It was shown to provide higher accuracy for classification than using all or only closed sequential patterns. Furthermore, mining generators is a key step in several other data mining tasks such as sequential rule generation. However, mining generators is computationally expensive. To address this issue, we propose a novel mining algorithm named *VGEN* (*Vertical sequential GENerator miner*). An experimental study on five real datasets shows that VGEN is up to two orders of magnitude faster than the state-of-the-art algorithms for sequential generator mining.

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

1 Introduction

Sequential pattern mining is a popular data mining task, which aims at discovering interesting patterns in sequences [10]. 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* called *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 [10]. Several algorithms have been proposed for sequential pattern mining such as *PrefixSpan* [11], *SPAM* [2], *SPADE* [15] and *CM-SPADE* [5]. 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 make a good analysis of 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. 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 [3, 4, 6, 13]. Two of those representations are *closed sequential patterns* [6, 13] and *sequential generator patterns* [8, 12] that allow us to keep all the information about all the frequent patterns that can be potentially generated.

Among the aforementioned representations, mining sequential generators is desirable for several reasons. First, if sequential generators are used with closed patterns, they can provide additional information that closed patterns alone cannot provide [12]. For example, a popular application of sequential generators is to generate sequential rules with a minimum antecedent and a maximum consequent [9]. When we focus on obtaining rules, by using generators as antecedents and closed patterns as consequents, we obtain rules which allows deriving the maximum amount of information based on the minimum amount of information [9]. Second, it was shown that generators provide higher classification accuracy and are more useful for model selection than using all patterns or only closed patterns [7, 12]. Third, sequential generators are preferable according to the principles of the MDL (Minimum Description Length) as they represent the smallest members of equivalence classes rather than the largest ones [8, 12]. Lastly, the set of sequential generators is compact. It has a similar size or can be smaller than the set of closed sequential patterns [8].

Although mining sequential generators is desirable, it remains computationally expensive and few algorithms have been proposed for this task. Most algorithms such as GenMiner [8], FEAT [7] and FSGP [14] employ a pattern-growth approach by extending the PrefixSpan algorithm [11]. These algorithms differ by how they store patterns, prune the search space, and whether they identify generators on-the-fly or by post-processing. Because these algorithms all adopts a pattern-growth approach, they also suffer from its main limitation which is to repeatedly perform database 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). Recently, an algorithm named MSGPs was also proposed. But it only provides a marginal performance improvement over previous approaches (up to 10%)[12].

In this paper, we propose a novel algorithm for mining sequential generators named *VGEN* (*Vertical sequential GENerator miner*). VGEN performs 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 and requires a single database scan to build its vertical structure. An experimental study with five real-life datasets shows that VGEN is up to two orders of magnitude faster than the state-of-the-art algorithms for sequential generator mining and performs very well on dense datasets.

The rest of the paper is organized as follows. Section 2 formally defines the problem of sequential generator 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.

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) $1, 2...p$.

Running example. Consider the following sequence database:

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

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 $s_a \sqsubseteq s_b$). In such a case, s_a is said to be a *sub-pattern* of s_b, and s_b to be a *super-pattern* of s_a **Example.** Sequence 4 of the running example 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.

Definition 4 (extensions). A sequence s_b is said to be an ***s*-extension** of a sequence $s_a = \langle I_1, I_2, ...I_h \rangle$ with an item x, iff $s_b = \langle I_1, I_2, ...I_h, \{x\} \rangle$, i.e. s_a is a prefix of s_b and the item x appears in an itemset later than all the itemsets of s_a. In the same way, the sequence s_c is said to be an ***i*-extension** of s_a with an item x, iff $s_c = \langle I_1, I_2, ...I_h \cup \{x\} \rangle$, i.e. s_a is a prefix of s_c and the item x occurs in the last itemset of s_a, and the item x is the last one in I_h, according to $\succ_{lex}$.

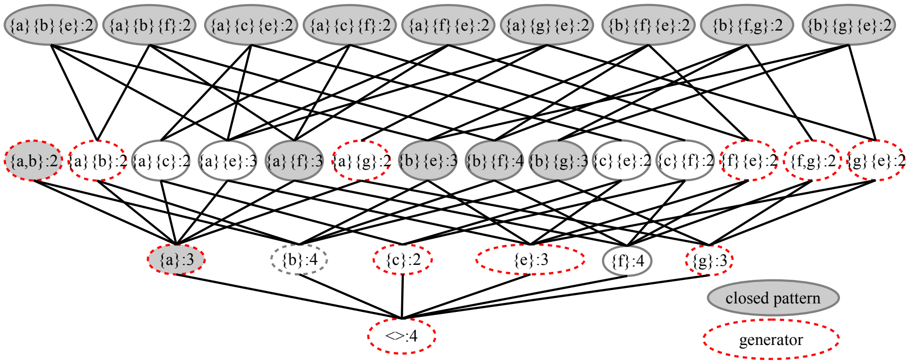


Fig. 1. All/closed/generator sequential patterns for the running example when $minsup = 2$

Definition 5 (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 6 (sequential pattern mining). Let $minsup$ be a minimum 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.** The lattice shown in Fig. 1 presents the 30 sequential patterns found in the database of the running example 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.

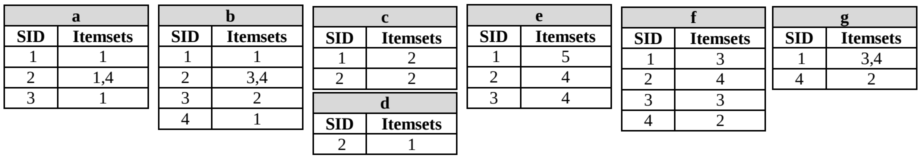
Definition 7 (closed/generator sequential pattern mining). A sequential pattern s_a is said to be *closed* if there is no other sequential pattern s_b, such that $s_a \sqsubseteq s_b$, and their supports are equal. A sequential pattern s_a is said to be a *generator* if there is no other sequential pattern s_b, such that $s_b \sqsubseteq s_a$, and their supports are equal. An alternative and equivalent definition is the following. Let an *equivalence class* be the set of all patterns supported by the same set of sequences, partially-ordered by the $\sqsubseteq$ relation. Generator (closed) patterns are the minimal (maximal) members of each equivalence class [8]. The *problem of mining closed (generator) sequential patterns* is to discover the set of closed (generator) sequential patterns. **Example.** Consider the database of the running example and $minsup = 2$. There are 30 sequential patterns (shown in Fig. 1), such that 15 are closed (identified by a gray color) and only 11 are generators (identified by a dashed line). It can be observed that in this case, the number of generators is less than the number of closed patterns. Consider the equivalence class $\{\langle\{c\}\rangle, \langle\{a\},\{c\}\rangle, \langle\{c\},\{e\}\rangle, \langle\{a\},\{c\},\{e\}\rangle, \langle\{a\},\{c\},\{f\}\rangle\}$ supported by sequences 1 and 2. In this equivalence class, pattern $\langle\{c\}\rangle$ is the only generator and $\langle\{a\},\{c\},\{e\}\rangle$ and $\langle\{a\},\{c\},\{f\}\rangle$ are the closed patterns.

We next introduce the main pruning property for sequential generator mining, which is used in various forms by state-of-the-art algorithms for generator mining to prune the search space [8].

Definition 8 (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 each sequence 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 $s_a \sqsubseteq s_b$, 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 $A_1 \subseteq B_{k1}$, $A_2 \subseteq B_{k2}$, ... $A_n \subseteq B_{km}$. Otherwise, the projection is undefined. **Example.** Consider the sequence database of the running example. The projection of the database by $\langle \{a\}, \{c\} \rangle$ is $\langle \langle \{f, g\}, \{g\}, \{e\} \rangle, \langle \{b\}, \{a, b, e, f\} \rangle \rangle$.

Property 1. (non-generator pruning). Let SDB be a sequence database, and s_a and s_b be two distinct sequential patterns. If $s_b \sqsubseteq s_a$ and $SDB_{s_a} = SDB_{s_b}$, then s_a and any extensions of s_a are not generators [8]. **Example.** For the running example, the projections of $\langle \{f\} \rangle$ and $\langle \{f, g\} \rangle$ are identical. Therefore, $\langle \{f, g\} \rangle$ and any of its extensions are not generators.

Definition 9 (horizontal/vertical database format). A *sequence database in horizontal format* is a database where each entry is a sequence. 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 [2]. **Example.** The sequence database of the running example was presented in horizontal format. Fig. 2 shows the same database but in vertical format.



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

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

c	
SID	**Itemsets**
1	2
2	2

d	
SID	**Itemsets**
2	1

e	
SID	**Itemsets**
1	5
2	4
3	4

f	
SID	**Itemsets**
1	3
2	4
3	3
4	2

g	
SID	**Itemsets**
1	3,4
4	2

Fig. 2. The vertical representation of the database from our running example

Vertical mining algorithms associate a structure named *IdList* [15, 2] 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 (see [15] for details). Several works proposed alternative representations for IdLists to save time in join operations, being the bitset representation the most efficient one [2].

3 The VGEN Algorithm

We present VGEN, our novel algorithm for sequential generator mining. It adopts the IdList structure implemented as bitsets [2, 15]. We first describe the general search procedure used by VGEN to explore the search space of sequential patterns [2]. Then, we describe how it is adapted to discover sequential generators efficiently.

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 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 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 [2, 15] 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]. 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 returns 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 so far. 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.

- *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 7) 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.

```
**PATTERN-ENUMERATION**(*SDB, minsup*)
Scan *SDB* to create *V*(*SDB*) and identify $S_{init}$, the list of frequent items.
**FOR** each item s ∈ $S_{init,}$
    **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*)
Output pattern *pat*.
$S_{temp} := I_{temp} := \emptyset$
**FOR** each item *j* ∈ $S_{n,}$
    **IF** the s-extension of *pat* is frequent **THEN** $S_{temp} := S_{temp} \cup \{i\}$.
**FOR** each item *j*∈ $S_{temp,}$
    **SEARCH**(the s-extension of *pat* with *j*, $S_{temp\ ,}$ elements in $S_{temp}$ greater than *j*, *minsup*).
**FOR** each item *j* ∈ $I_{n,}$
    **IF** the i-extension of *pat* is frequent **THEN** $I_{temp} := I_{temp} \cup \{i\}$.
**FOR** each item *j* ∈ $I_{temp,}$
    **SEARCH**(i-extension of *pat* with *j*, $S_{temp\ ,}$ all elements in $I_{temp}$ greater than *j*, *minsup*).
```

Fig. 3. The search procedure

- *Super-pattern checking.* If s_a is determined to be a generator according to sub-pattern checking, 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 7) 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 four 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 are 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 $s_a \sqsubseteq s_b$ ($s_b \sqsubseteq s_a$).

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 as prefix, thus indicating that x is not a generator).

This optimization allows to avoid some sub-pattern checks. However it does allows the algorithm to prune the search space of frequent patterns to avoid considering patterns that are non generators. To prune the search space, we add a pruning condition based on Property 1. During sub-pattern checking for a pattern x, if a smaller pattern y can be found in Z such that the projected database is identical, then x is not a generator as well as any extension of x. Therefore, extensions of x should not be explored (by Property 1). Checking if projected databases are identical is done 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) [5] 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 $j \in I$ succeeding 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 the running example 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. Let a sequential pattern *pat* being considered for *s*-extension (*i*-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 $x \in cm_s(a)$ $(x \in cm_i)$, 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 *s*-extension (*i*-extension) having *pat* as prefix, by not adding x to the variable S_{temp} (I_{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.

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 hashsets, 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 $minsup = 2$.

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 FSGP [14] and FEAT [7] for mining sequential generators and BIDE [13] for closed sequential pattern mining. Note that we do not compare with MSGPs for generator mining since it only provide a marginal speed improvement over FSGP (up to 10%) [12]. Furthermore, we do not compare with GenMiner, since authors of GenMiner reported that it is slower than BIDE [8]. All memory measurements were done using the Java API. Experiments were carried on five real-life datasets commonly used in the data mining literature, having varied characteristics and representing three different types of data (web click stream, text from a book and protein sequences). Table 4 summarizes the datasets' characteristics. The source code of all compared algorithms and datasets can be downloaded from `http://goo.gl/xat4k`.

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, memory usage and pattern count. Note that the execution time of VGEN includes the creation of ID-Lists.

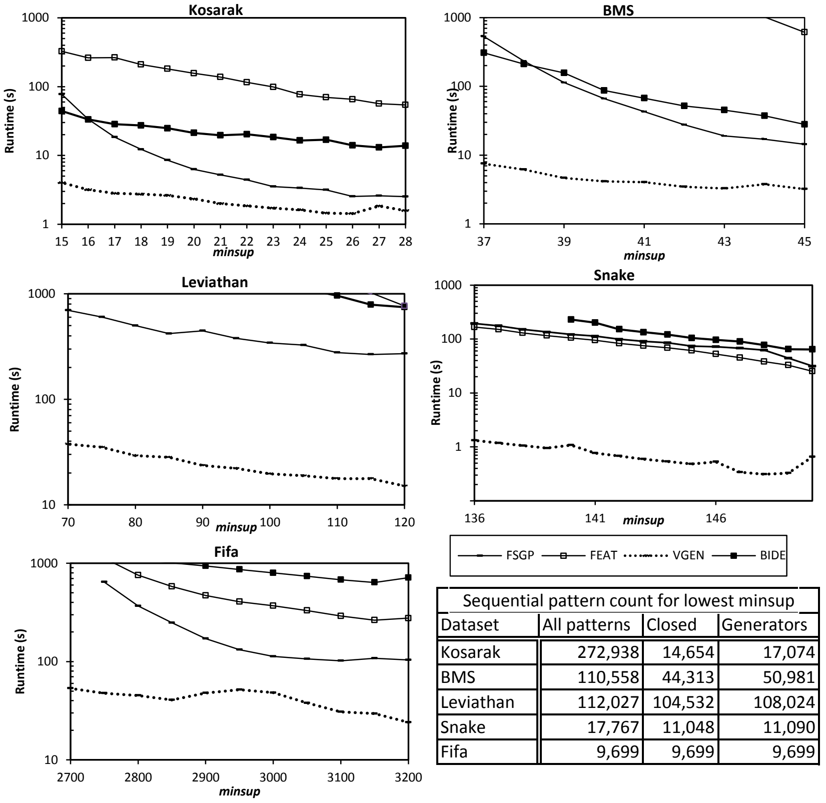
In terms of execution time, results (see Fig. 4) show that VGEN is from one to two orders of magnitude faster than FEAT, FSGP and BIDE for all datasets. Furthermore, we observe that VGEN has excellent performance on dense datasets. For example, on Snake, VGEN is 127 times faster than FEAT, 149 times faster than FSGP, and 263 times faster than BIDE.

In terms of pattern count (see Fig. 4), as expected, we observe that the set of sequential generators can be much smaller than the set of all sequential patterns, and that there is about as much sequential generators as closed patterns.

In terms of memory consumption the maximum memory usage in megabytes of VGEN/FSGP/FEAT/BIDE for the Kosarak, BMS, Leviathan, Snake and FIFA datasets for the lowest *minsup* value were respectively 820/840/381/427, 1641/759/*/383, 3422/1561/*/*, 135/795/776/329 and 1815/1922/1896/2102, where * indicates that an algorithm has no result because it ran out of memory or failed to execute within the time limit of 1000s. We note that VGEN has the lowest memory consumption for Snake and FIFA, the second lowest for Leviathan and the third lowest for BMS and Kosarak.

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.

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



Sequential pattern count for lowest minsup			
Dataset	All patterns	Closed	Generators
Kosarak	272,938	14,654	17,074
BMS	110,558	44,313	50,981
Leviathan	112,027	104,532	108,024
Snake	17,767	11,048	11,090
Fifa	9,699	9,699	9,699

Fig. 4. Execution times and pattern count

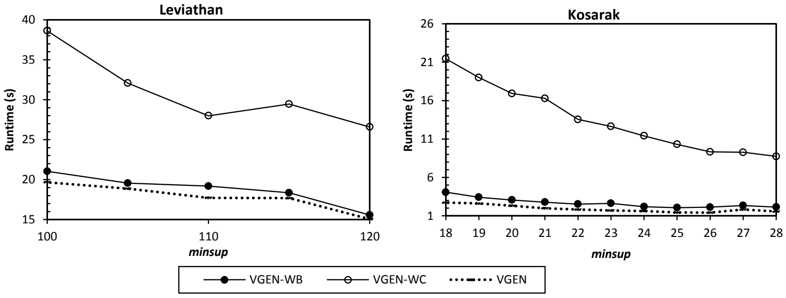


Fig. 5. Influence of optimizations for Leviathan (left) and Kosarak (right)

5 Conclusion

In this paper, we presented a depth-first search algorithm for mining sequential generators named VGEN (Vertical sequential GENerator miner). It relies on a vertical representation of the database and includes three novel strategies named

ENG (Efficient filtering of Non-Generator patterns), BEC (Backward Extension checking) and CPC (Candidate Pruning by Co-occurrence map) to efficiently identify generators and prune the search space. We performed an experimental study with five real-life datasets to evaluate the performance of VGEN. Results show that VGEN is up to two orders of magnitude faster than the state-of-the-art algorithms for sequential generator mining. The source code of VGEN and all compared algorithms can be downloaded from `http://goo.gl/xat4k`.

Acknowledgement. This work is financed by a National Science and Engineering Research Council (NSERC) of Canada research grant.

References

1. Agrawal, R., Ramakrishnan, S.: Mining sequential patterns. In: Proc. 11th Intern. Conf. Data Engineering, pp. 3–14. IEEE (1995)
2. 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)
3. 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, pp. 169-180 (2013)
4. Fournier-Viger, P., Wu, C.-W., Gomariz, A., Tseng, V. S.: VMSP: Efficient Vertical Mining of Maximal Sequential Patterns. Proc. 27th Canadian Conference on Artificial Intelligence, pp. 83–94 (2014)
5. Fournier-Viger, P., Gomariz, A., Campos, M., Thomas, R.: Fast Vertical Sequential Pattern Mining Using Co-occurrence Information. In: Proc. 18th Pacific-Asia Conf. Knowledge Discovery and Data Mining, pp. 40–52 (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 (2013)
7. 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)
8. 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)
9. Lo, D., Khoo, S.-C., Wong, L.: Non-redundant sequential rules: Theory and algorithm. Information Systems 34(4), 438–453 (2011)
10. Mabroukeh, N.R., Ezeife, C.I.: A taxonomy of sequential pattern mining algorithms, ACM Computing Surveys 43(1), 1–41 (2010)
11. 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)
12. 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)
13. Wang, J., Han, J., Li, C.: Frequent closed sequence mining without candidate maintenance. IEEE Trans. on Knowledge Data Engineering 19(8), 1042–1056 (2007)
14. Yi, S., Zhao, T., Zhang, Y., Ma, S., Che, Z.: An effective algorithm for mining sequential generators. Procedia Engineering, 15, 3653-3657 (2011)
15. Zaki, M.J.: SPADE: An efficient algorithm for mining frequent sequences. Machine Learning 42(1), 31–60 (2001)