Computer and Information Science; Vol. 10, No. 1; 2017
ISSN 1913-8989 E-ISSN 1913-8997
Published by Canadian Center of Science and Education

The Impact of the Pattern-Growth Ordering on the Performances of Pattern Growth-Based Sequential Pattern Mining Algorithms

Kenmogne Edith Belise[1]

[1] Faculty of Science, Department of Mathematics and Computer Science, Cameroon

Correspondence: Kenmogne Edith Belise, Faculty of Science, Department of Mathematics and Computer Science, LIFA, Po. Box. 67 Dschang, Cameroon. Tel: 237-677-7496-61. E-mail: ebkenmogne@gmail.com

Received: October 15, 2015 Accepted: November 25, 2015 Online Published: December 25, 2016

doi:10.5539/cis.v10n1p23 URL: http://dx.doi.org/10.5539/cis.v10n1p23

Abstract

Sequential Pattern Mining is an efficient technique for discovering recurring structures or patterns from very large dataset widely addressed by the data mining community, with a very large field of applications, such as cross-marketing, DNA analysis, web log analysis, user behavior, sensor data, etc. The sequence pattern mining aims at extracting a set of attributes, shared across time among a large number of objects in a given database. Previous studies have developed two major classes of sequential pattern mining methods, namely, the candidate generation-and-test approach based on either vertical or horizontal data formats represented respectively by GSP and SPADE, and the pattern-growth approach represented by FreeSpan and PrefixSpan. In this paper, we are interested in the study of the impact of the pattern-growth ordering on the performances of pattern growth-based sequential pattern mining algorithms. To this end, we introduce a class of pattern-growth orderings, called linear orderings, for which patterns are grown by making grow either the current pattern prefix or the current pattern suffix from the same position at each growth-step. We study the problem of pruning and partitioning the search space following linear orderings. Experimentations show that the order in which patterns grow has a significant influence on the performances.

Keywords: sequence mining, sequential pattern, pattern-growth direction, pattern-growth ordering, search space, pruning, partitioning

1. Introduction

A sequence database consists of sequences of ordered elements or events, recorded with or without a concrete notion of time. Sequences are common, occurring in any metric space that facilitates either partial or total ordering. Customer transactions, codons or nucleotides in an amino acid, website traversal, computer networks, DNA sequences and characters in a text string are examples of where the existence of sequences may be significant and where the detection of frequent (totally or partially ordered) subsequences might be useful. Sequential pattern mining has arisen as a technology to discover such subsequences. A subsequence, such as buying first a PC, then a digital camera, and then a memory card, if it occurs frequently in a customer transaction database, is a (frequent) sequential pattern.

Sequential pattern mining (Dam et al., 2016; Mabroukeh & Ezeife, 2010; Lin et al., 2016a; Linet al., 2016b; Lin et al., 2016c; Lin et al., 2016d) is an important data mining problem widely addressed by the data mining community, with a very large field of applications such as finding network alarm patterns, mining customer purchase patterns, identifying outer membrane proteins, automatically detecting erroneous sentences, discovering block correlations in storage systems, identifying plan failures, identifying copy-paste and related bugs in large-scale software code, API specification mining and API usage mining from open source repositories, and Web log data mining. Sequential pattern mining aims at extracting a set of attributes, shared across time among a large number of objects in a given database.

The sequential pattern mining problem was first introduced by Agrawal & Srikant (1995) based on their study of customer purchase sequences, as follows: *Given a set of sequences, where each sequence consists of a list of events (or elements) and each event consists of a set of items, and given a user-specified minimum support threshold min_sup, sequential pattern mining finds all frequent subsequences, that is, the subsequences whose occurrence frequency in the set of sequences is no less than min_sup.*

In this paper, we are interested in the study of the impact of the pattern-growth ordering on the performances of pattern growth-based sequential pattern mining algorithms. It aims at enhancing understanding of the pattern-growth approach. To this end, the important key concepts upon which that approach relies, namely pattern-growth direction, pattern-growth ordering, search space pruning and search space partitioning, are revisited. We introduce a class of pattern-growth orderings, called linear orderings, for which patterns are grown by making grow either the current pattern prefix or the current pattern suffix from the same position at each growth-step. This class contains PrefixSpan (Pei et al., 2001; Pei et al., 2004) and involves both unidirectional and bidirectional growth. Thus, it is a generalization of PrefixSpan (Pei et al., 2001; Pei et al., 2004). However, it does not contain FreeSpan (Han et al., 2000) as it makes grow patterns from any position. We study the problem of pruning and partitioning the search space following linear orderings. Experimentations show that the order in which patterns grow has a significant influence on the performances.

The rest of the paper is organized as follows. Section 2 presents the formal definition of the problem of sequential pattern mining. Section 3 presents previous results. Section 4 presents the theoretical contribution of the paper. Section 5 presents experimental results. Concluding remarks are given in section 6.

2. Problem statement and Notation

The problem of mining sequential patterns, and its associated notation, can be given as follows: Let I={i_1, i_2, . . . , i_n} be a set of literals, termed **items**, which comprise the alphabet. An **itemset** is a subset of items. A **sequence** is an ordered list of itemsets. Sequence s is denoted by $\langle s_1, s_2, \ldots s_n\rangle$, where s_j is an itemset. s_j is also called an **element** of the sequence, and denoted as $(x_1, x_2,\ldots,x_m)$, where x_k is an item. For brevity, the brackets are omitted if an element has only one item, i.e. element (x) is written as x. An item can occur at most once in an element of a sequence, but can occur multiple times in different elements of a sequence. The number of instances of items in a sequence is called the length of the sequence. A Sequence with length l is called an **l-sequence**. The length of a sequence α is denoted $|\alpha|$. A sequence $\alpha=\langle a_1\ a_2 \ldots a_n\rangle$, is called **subsequence** of another sequence $\beta=\langle b_1\ b_2 \ldots b_m\rangle$ and β a **supersequence** of α, denoted as $\alpha \subseteq \beta$, if there exist integers $1 \leq j_1 < j_2 < \ldots < j_n \leq j_m$ such that $a_1 \subseteq b_{j1}$, $a_2 \subseteq b_{j2}$, … $a_n \subseteq b_{jn}$. Symbol ε denotes the **empty sequence**.

We are given a database S of input-sequences. A **sequence database** is a set of tuples of the form ⟨sid, s⟩ where sid is a **sequence_id** and s a sequence. A tuple ⟨sid, s⟩ is said to contain a sequence α if α is a subsequence of s. The **support** of a sequence α in a sequence database S is the number of tuples in the database containing α, i.e.

$$\text{support}(S, \alpha) = |\{\langle sid, s\rangle \mid \langle sid, s\rangle \in S \text{ and } \alpha \subseteq s\}|.$$

It can be denoted as support(α) if the sequence database is clear from the context. Given a user-specified positive integer denoted min_support, termed the **minimum support** or the **support threshold**, sequence α is called a **sequential pattern** in the sequence database S if support(S,α)≥ min_support. A sequential pattern with length l is called an **l-pattern**. Given a sequence database and the min_support threshold, **sequential pattern mining** is to find the complete set of sequential patterns in the database.

3. Related work

Sequential pattern mining is an important data mining problem. Since the first proposal of this data mining task and its associated efficient mining algorithms, there has been a growing number of researchers in the field and tremendous progress (Mabroukeh & Ezeife, 2010) has been made, evidenced by hundreds of follow-up research publications, on various kinds of extensions and applications, ranging from scalable data mining methodologies, to handling a wide diversity of data types, various extended mining tasks, and a variety of new applications.

Improvements in sequential pattern mining algorithms have followed similar trend in the related area of association rule mining and have been motivated by the need to process more data at a faster speed with lower cost. Previous studies have developed two major classes of sequential pattern mining methods : Apriori-based approaches (Agrawal & Srikant, 1995; Ayres et al., 2002; Garofalakis et al., 1999; Gouda et al., 2007; Gouda et al., 2010; Masseglia et al., 1998; Savary & Zeitouni, 2005; Yang & Kitsuregawa, 2005; Zaki, 2000; Zaki, 2001) and pattern growth algorithms (Han et al., 2000; Pei et al., 2000; Pei et al., 2001; Pei et al., 2004; Hsieh et al., 2008; Seno & Karypis, 2008).

The Apriori-based approach form the vast majority of algorithms proposed in the literature for sequential pattern mining. Apriori-like algorithms depend mainly on the Apriori anti-monotony property, which states the fact that any super-pattern of an infrequent pattern cannot be frequent, and are based on a candidate generation-and-test paradigm proposed in association rule mining (Agrawal et al., 1993; Agrawal & Srikant, 1994). This candidate generation-and-test paradigm is carried out by GSP (Agrawal & Srikant, 1995), SPADE (Zaki, 2001), and SPAM (Ayres et al., 2002). Mining algorithms derived from this approach are based on either vertical or horizontal data formats. Algorithms based on the vertical data format involve AprioriAll, AprioriSome and DynamicSome (Agrawal & Srikant, 1995), GSP (Agrawal & Srikant, 1995), PSP (Masseglia et al., 1998) and SPIRIT (Garofalakis et al., 1999), while those based on the horizontal data format involve SPADE (Zaki, 2001), cSPADE (Zaki, 2000), SPAM (Ayres et al., 2002), LAPIN-SPAM (Yang & Kitsuregawa, 2005), IBM (Savary & Zeitouni, 2005) and PRISM (Gouda et al., 2007; Gouda et al., 2010) . The generation-and-test paradigm has the disadvantage of repeatedly generating an explosive number of candidate sequences and scanning the database to maintain the support count information for these sequences during each iteration of the algorithm, which makes them computationally expensive. To increase the performance of these algorithms constraint driven discovery can be carried out. With constraint driven approaches systems should concentrate only on user specific or user interested patterns or user specified constraints such as minimum support, minimum gap or time interval etc. With regular expressions these constraints are studied in SPIRIT (Garofalakis et al., 1999).

To alleviate these problems, the pattern-growth approach, represented by FreeSpan (Han et al., 2000), PrefixSpan (Pei et al., 2001; Pei et al., 2004) and their further extensions, namely FS-Miner (El-Sayed et al., 2004), LAPIN (Hsieh et al., 2008 ; Yang et al., 2007), SLPMiner (Seno & Karypis, 2002) and WAP-mine (Pei et al., 2000), for efficient sequential pattern mining adopts a divide-and-conquer pattern growth paradigm as follows. Sequence databases are recursively projected into a set of smaller projected databases based on the current sequential patterns, and sequential patterns are grown in each projected database by exploring only locally frequent fragments (Han et al., 2000; Pei et al., 2004). The frequent pattern growth paradigm removes the need for the candidate generation and prune steps that occur in the Apriori-based algorithms and repeatedly narrows the search space by dividing a sequence database into a set of smaller projected databases, which are mined separately. The major advantage of projection-based sequential pattern-growth algorithms is that they avoid the candidate generation and prune steps that occur in the Apriori-based algorithms. Unlike Apriori-based algorithms, they grow longer sequential patterns from the shorter frequent ones. The major cost of these algorithms is the cost of forming projected databases recursively. To alleviate this problem, a pseudo-projection method is exploited to reduce this cost. Instead of performing physical projection, one can register the index (or identifier) of the corresponding sequence and the starting position of the projected suffix in the sequence. Then, a physical projection of a sequence is replaced by registering a sequence identifier and the projected position index point. Pseudo-projection reduces the cost of projection substantially when the projected database can fit in main memory.

PrefixSpan (Pei et al., 2001; Pei et al., 2004) and FreeSpan (Han et al., 2000) differ at the criteria of partitioning projected databases and at the criteria of growing patterns. FreeSpan (Han et al., 2000) creates projected databases based on the current set of frequent patterns without a particular ordering (i.e., pattern-growth direction), whereas PrefixSpan projects databases by growing frequent prefixes. Thus, PrefixSpan follows the unidirectional growth whereas FreeSpan follows the bidirectional growth. Another difference between FreeSpan and PrefixSpan is that the pseudo-projection works efficiently for PrefixSpan but not so for FreeSpan. This is because for PrefixSpan, an offset position clearly identifies the suffix and thus the projected subsequence. However, for FreeSpan, since the next step pattern-growth can be in both forward and backward directions from any position, one needs to register more information on the possible extension positions in order to identify the remainder of the projected subsequences.

4. Proposed Work

4.1. Pattern-Growth Directions and Orderings

Definition 1 (*Pattern-growth direction*). A pattern-growth direction is a direction along which patterns could grow. There are two pattern-growth directions, namely *left-to-right* and *right-to-left* directions. Do grow a pattern along *left-to-right* (resp. *right-to-left*) direction is to add one or more item to its right (resp. left) hand side.

Definition 2 (*Pattern-growth ordering*). A pattern-growth ordering is a specification of the order in which patterns should grow. A pattern-growth ordering is said to be unidirectional iff all the patterns should grow along a unique direction. Otherwise it is said to be bidirectional. A pattern-growth ordering is said to be static (resp. dynamic) iff it is fully specified before the beginning of the mining process (resp. iff it is constructed during the mining process).

Definition 3 (*Basic-static pattern-growth ordering*). A basic-static pattern-growth ordering, also called basic pattern-growth ordering for sake of simplicity, is an ordering which is based on a unique pattern-growth direction, and grow a pattern at the rate of one item per growth-step.

There are two basic-static pattern-growth orderings, namely *left-to-right ordering* (also called *prefix-growth ordering*), which consists in growing a prefix of a pattern at the rate of one item per growth-step at its right hand side, and *right-to-left ordering* (also called *suffix-growth ordering*), which consists in growing a suffix of a pattern at the rate of one item per growth-step at its left hand side.

Definition 4 (*Basic-dynamic pattern-growth ordering*). A basic-dynamic pattern-growth ordering is an ordering which grow a pattern at the rate of one item per growth-step, and whose pattern-growth direction is determined at the beginning of each growth-step during the mining process. It is denoted $*$-growth.

Definition 5 (*Basic-bidirectional pattern-growth ordering*). A basic-bidirectional pattern-growth ordering is an ordering which is based on the two distinct pattern-growth directions, and grow a pattern in each direction at the rate of one item per couple of growth-steps.

There are two basic-bidirectional pattern-growth orderings, namely *prefix-suffix-growth ordering* (i.e. *left-to-right direction* followed by *right-to-left direction*), which consists in growing a pattern at the rate of one item per growth-step during a couple of steps by first growing a prefix (i.e. adding of one item at the right-hand side) of that pattern followed by the growing of the corresponding suffix (i.e. adding of one item at the left-hand side), and *suffix-prefix-growth ordering* (i.e. *right-to-left direction* followed by *left-to-right direction*), which consists in growing a pattern at the rate of one item per growth-step during a couple of steps by first growing a suffix of that pattern followed by the growing of the corresponding prefix.

Definition 6 (*Linear pattern-growth ordering*). A linear pattern-growth ordering is a series of compositions of *$*$-growth*, *prefix-growth* and *suffix-growth* orderings, and denoted o_0-o_1-o_2 … o_{n-1}-growth for some n, where $o_i \in$ {prefix, suffix, *} ($0 \le i \le n-1$). It is said to be static iff $o_i \in$ {prefix, suffix} for all $i \in \{0, 1, 2, …, n-1\}$. Otherwise, it is said to be dynamic.

The o_0-o_1-o_2 … o_{n-1}-growth linear ordering consists in growing a pattern at the rate of one item per growth-step during a series of n growth-steps by growing at step i ($0 \le i \le n-1$) a prefix (resp. suffix) of that pattern if o_i denotes prefix (resp. suffix). If $o_i \in \{*\}$, a pattern-growth direction is determined and an item is added to the pattern following that direction. For instance, stemming from the *prefix-suffix-suffix-prefix-growth* static linear ordering, one should grow a pattern in the following order:

- *Growth-step 0*: Add an item to the right hand side of a prefix of that pattern.
- *Growth-step 1*: Add one item to the left hand side of the corresponding suffix of the previous prefix.
- *Growth-step 2*: Repeat step 1.
- *Growth-step 3*: Repeat step 0.
- *Growth-step k (k ≥4)*: Repeat step *k mod 4.*

The *prefix-suffix-$*$-prefix-growth* dynamic linear ordering grows patterns as *prefix-suffix-suffix-prefix-growth* ordering except for steps k that satisfy (k mod 4) = 3. During such a particular step, a pattern-growth direction is determined and an item is added to the pattern following that direction.

FreeSpan and PrefixSpan differ at the criteria of growing patterns. FreeSpan creates projected databases based on the current set of frequent patterns without a particular ordering (i.e., pattern-growth direction). Since a length-k pattern may grow at any position, the search for length-(k+1) patterns will need to check every possible combination, which is costly. Because of this, FreeSpan do not follow the linear ordering. However PrefixSpan follows the prefix-growth static ordering as it projects databases by growing frequent prefixes.

Given a database of sequences, an open problem is to find a linear ordering that leads to the best mining performances over all possible linear orderings.

4.2 Search Space Pruning and Partitioning

Definition 7 (*Prefix of an itemset*). Suppose all the items within an itemset are listed alphabetically. Given an itemset $x = (x_1x_2 … x_n)$, another itemset $x' = (x'_1x'_2 … x'_m)$ ($m \le n$) is called a prefix of x if and only if $x'_I = x_i$ for all $i \le m$. If $m < n$, the prefix is also denoted as $x = (x_1x_2... x_m\_)$.

Definition 8 (*The corresponding suffix of a prefix of an itemset*). Let $x = (x_1x_2 … x_n)$ be a itemset. Let $x' = (x_1x_2 … x_m)$ ($m \le n$) be a prefix of x. Itemset $x'' = (x_{m+1}x_{m+2}… x_n)$ is called the suffix of x with regards to prefix x′, denoted as $x'' = x/x'$. We also denote $x = x'.x''$. Note, if $x = x'$, the suffix of x with regards to x′ is empty. If $1 \le m < n$, the suffix is also denoted as $(\_x_{m+1}x_{m+2} … x_n)$.

For example, for the itemset iset=(abcdefgh),(_efgh) is the suffix with regards to the prefix (abcd_), iset=(abcd_).(_efgh), (abcdef_) is the prefix with regards to suffix (_gh) and iset=(abcdef_).(_gh).

The following definition introduces the dot operator. It permits itemset concatenations and sequence concatenations.

Definition 9 (*"." operator*). Let e and e′ be two itemsets that do not contain the underscore symbol (_). Assume that all the items in e′ are alphabetically sorted after those in e. Let $\gamma=\langle e_1 \dots e_{n-1}a\rangle$ and $\mu=\langle be'_2 \dots e'_m\rangle$ be two sequences, where e_i and e'_i are itemsets that do not contain the underscore symbol, a ∈ {e, (_items in e), (items in e_), (_items in e_)} and b ∈ {e′, (_items in e′), (items in e′_), (_items in e′_)}. The dot operator is defined as follows.

1. e . e′ = ee′
2. e . (_items in e′) = (items in e ∪ e′)
3. e . (items in e′_) = e (items in e′_)
4. e . (_items in e′_)= (items in e ∪ e′_)
5. (items in e_) . e′ = (items in e ∪ e′)
6. (items in e_) . (_items in e′) = (items in e ∪ e′)
7. (items in e_) . (_items in e′_)= (items in e ∪ e′_)
8. (items in e_) . (items in e′_)= (items in e ∪ e′_)
9. (_items in e) . e′ = (_items in e)e′
10. (_items in e) . (items in e′_) = (_items in e)(items in e′_)
11. (_items in e) . (_items in e′_)= (_items in e ∪ e′_)
12. (_items in e) . (_items in e′)= (_items in e ∪ e′)
13. (_items in e_) . e′ = (_items in e ∪ e′)
14. (_items in e_) . (_items in e′_) = (_items in e ∪ e′_)
15. (_items in e_) . (items in e′_)=(_items in e ∪ e′_)
16. (_items in e_) . (_items in e′)= (_ items in e ∪ e′)
17. $\gamma.\mu = \langle e_1 \dots e_{n-1}a.be'_2 \dots e'_m\rangle$

For example, s=⟨a(abc)(ac)(efgh)⟩=⟨(a).(a_).(_b_).(_c).(a_).(_c).(e_).(_f_).(_g_).(_h)⟩ and s=⟨(a)⟩.⟨(a_)⟩.⟨(_b_)⟩.⟨(_c)⟩.⟨(a_)⟩.⟨(_c)⟩.⟨(e_)⟩.⟨(_f_)⟩.⟨(_g_)⟩.⟨(_h)⟩.

Definition 10 (*Prefix of a sequence*) (Pei et al., 2004). Suppose all the items within an element are listed alphabetically. Given a sequence $\alpha=\langle e_1e_2 \dots e_n\rangle$, a sequence $\beta=\langle e'_1e'_2 \dots e'_m\rangle$ ($m \le n$) is called a prefix of α if and only if 1) $e'_I=e_i$ for all $i \le m-1$; 2) $e'_m \subseteq e_m$; and 3) all the frequent items in e_m - e'_m are alphabetically sorted after those in e'_m. If $e'_m\neq\emptyset$ and $e'_m \subseteq e_m$ the prefix is also denoted as $\langle e'_1e'_2 \dots e'_{m-1}$(items in e'_m_)⟩.

Definition 11 (*The corresponding suffix of a prefix of a sequence*) (Pei et al., 2004). Given a sequence $\alpha=\langle e_1e_2 \dots e_n\rangle$. Let $\beta=\langle e_1e_2 \dots e_{m-1}e'_m\rangle$ ($m \le n$) be a prefix of α. Sequence $\gamma=\langle e''_me_{m+1} \dots e_n\rangle$ is called the suffix of α with regards to prefix β, denoted as γ= α/β, where $e''_m = e_m - e'_m$. We also denote α=β.γ. Note, if β=α, the suffix of α with regards to β is empty. If e''_m is not empty, the suffix is also denoted as ⟨(_items in e''_m) $e_{m+1} \dots e_n$⟩.

For example, for the sequence s=⟨a(abc)(ac)(efgh)⟩, ⟨(ac)(efgh)⟩ is the suffix with regards to the prefix ⟨a(abc)⟩, ⟨(_bc)(ac)(efgh)⟩ is the suffix with regards to the prefix ⟨aa⟩, ⟨(_c)(ac)(efgh)⟩ is the suffix with regards to the prefix ⟨a(ab)⟩, and ⟨a(abc)(a_)⟩ is the prefix with regards to the suffix ⟨(_c)(efgh)⟩.

Given three sequences, y, α and α′, we denote spc(y,α) (resp. ssc(y,α′)) the shortest prefix (resp. suffix) of y containing α (resp. α′). If no prefix (resp. suffix) of y contains α (resp. α′) spc(y,α) (resp. ssc(y,α′)) does not exist. If the two sequences spc(y,α) and ssc(y,α′) exist and do not overlap in sequence y, there exists a sequence $y_{\alpha,\alpha'}$ such that y=spc(y,α).$y_{\alpha,\alpha'}$.ssc(y,α′). Hence, we have the following definition.

Definition 12 (*Canonical sequence decomposition*). Given three sequences, y, α and α′ such that spc(y,α) and ssc(y,α′) exist and do not overlap in y. Equation y=spc(y,α).$y_{\alpha,\alpha'}$.ssc(y,α′) is the canonical decomposition of y following prefix α and suffix α′. The left, middle and right parts of the decomposition are respectively spc(y,α), $y_{\alpha,\alpha'}$ and ssc(y,α′).

For example, consider sequence s=⟨a(abc)(ac)(efgh)⟩, we have spc(s,⟨a⟩)=⟨a⟩, spc(s,⟨(ab)⟩)=⟨a(ab)⟩, spc(s,⟨(ac)⟩)= ⟨a(abc)⟩, ssc(s,⟨(c)(e)⟩)=⟨(c)(efgh)⟩, ssc(s, ⟨a⟩)=⟨(ac)(efgh)⟩,ssc(s,⟨(bc)⟩)=⟨(_bc)(ac)(efgh)⟩, s=spc(s,⟨(ab)⟩).⟨(_c)(a_) ⟩.ssc(s,⟨(c)(e)⟩) and s=spc(s,⟨(ac)⟩).ε.ssc(s,⟨a⟩). The two sequences spc(s,⟨(ab)⟩) and spc(s,⟨(ab)⟩ overlap in sequence s as two sets of the index positions of their items in s are not disjoint.

Stemming from the canonical decompositions of sequences following prefix α and suffix α', we define two sets of the sequence database S as follows. We denote $S_{\alpha,\alpha}$ the set of subsequences of S prefixed with α and suffixed with α' which are obtained by replacing the left and right parts of canonical decompositions respectively with α and α'. We have $S_{\alpha,\alpha'}=\{\langle sid, \alpha.y_{\alpha,\alpha'}.\alpha'\rangle \mid \langle sid, y\rangle \in S \text{ and } y=spc(y,\alpha).y_{\alpha,\alpha'}.ssc(y,\alpha')\}$. We denote $S^{\alpha,\alpha'}$ the set of subsequences which are obtained by removing the left and right parts of canonical decompositions. We have $S^{\alpha,\alpha'}=\{\langle sid, y_{\alpha,\alpha'}\rangle \mid \langle sid, y\rangle \in S \text{ and } y=spc(y,\alpha).y_{\alpha,\alpha'}.ssc(y,\alpha')\}$. We also have $S=S_{\varepsilon,\varepsilon}$ and $S=S^{\varepsilon,\varepsilon}$ as ε denotes the empty sequence.

Definition 13 (*Extension of the "." operator*). Let S be a sequence database and let α be a sequence that may contain the underscore symbol (_). The dot operator is extended as follows. We have $\alpha.S=\{\langle sid,\alpha.s\rangle \mid \langle sid,s\rangle \in S\}$ and $S.\alpha=\{\langle sid,s.\alpha\rangle \mid \langle sid,s\rangle \in S\}$.

Corollary 1 (*Associatively of the "." operator*). The dot operator is associative, i.e. given a sequence database S and three sequences α, $\alpha'$and α'' that may contain the underscore symbol (_), we have:

1. $(\alpha.\alpha').\alpha'' = \alpha.(\alpha'.\alpha'')$
2. $\alpha.(\alpha'.S)=(\alpha.\alpha')$.
3. $(S.\alpha).\alpha'=S.(\alpha.\alpha')$
4. $(\alpha.S).\alpha' = \alpha.(S.\alpha')$

Proof. It is straightforward from the dot operation definition.

We have the following lemmas.

Lemma 1 (*The support of z in $S^{\alpha,\alpha'}$ is that of its counterpart in S*). Given a sequence database S and two sequences α and α', for any sequence y prefixed with α and suffixed with α', i.e. $y=\alpha.z.\alpha'$ for some sequence z, we have $support(S,y)=support(S^{\alpha,\alpha'},z)$.

Proof. Consider the function f from dataset $S_{\alpha,\alpha'}$ to dataset $S^{\alpha,\alpha'}$ which assigns tuple $\langle sid, y_{\alpha,\alpha'}\rangle \in S^{\alpha,\alpha'}$ to tuple $\langle sid, spc(y,\alpha).y_{\alpha,\alpha'}.ssc(y,\alpha')\rangle \in S_{\alpha,\alpha'}$ where tuple $\langle sid,y\rangle \in S$ and sequence y admits a canonical decomposition following prefix α and suffix α'.

Let's prove that function f is injective. Consider two tuples of S, $\langle sid, y\rangle$ and $\langle sid',y'\rangle$, each having a canonical decomposition following prefix α and suffix α'.

Assume that $f(\langle sid, spc(y,\alpha).y_{\alpha,\alpha'}.ssc(y,\alpha')\rangle)=f(\langle sid', spc(y',\alpha).y'_{\alpha,\alpha'}. ssc(y',\alpha')\rangle)$. This implies that $\langle sid,y_{\alpha,\alpha'}\rangle= \langle sid',y'_{\alpha,\alpha'}\rangle$, which in turn implies that sid=sid'. This implies that tuple $\langle sid,y\rangle$ is equal to $\langle sid',y'\rangle$ as the identifier of any tuple is unique. It comes that $y=y'$. Thus $\langle sid, spc(y,\alpha).y_{\alpha,\alpha'}.ssc(y,\alpha')\rangle=\langle sid', spc(y',\alpha).y'_{\alpha,\alpha'}. ssc(y',\alpha')\rangle$. Therefore function f is injective.

Let's prove that function f is surjective. Consider $\langle sid, z_{\alpha,\alpha'}\rangle \in S^{\alpha,\alpha'}$, where $\langle sid,z\rangle$ belongs to S and admits a canonical decomposition following prefix α and suffix α'. From the definition of function f, $f(\langle sid, spc(z,\alpha).z_{\alpha,\alpha'}. ssc(z,\alpha')\rangle)=\langle sid,z_{\alpha,\alpha'}\rangle$. This means that $\langle sid,z_{\alpha,\alpha'}\rangle \in S^{\alpha,\alpha'}$ admits a pre-image in $S_{\alpha,\alpha'}$. Thus function f is surjective.

Function f is bijective because it is injective and surjective. Let consider a sequence y prefixed with α and suffixed with α', i.e. $y=\alpha.z.\alpha'$ for some sequence z. Denote $S(y)=\{\langle sid,s\rangle \mid \langle sid,s\rangle \in S \text{ and } y \subseteq s\}$. Recall that $support(S,y)=|S(y)|$. The definition of S(y) means that it is the set of sequences of S having a canonical decomposition following prefix α and suffix α' and containing sequence z in their middle part. It comes that $S(y)=\{\langle sid,s\rangle \mid \langle sid,s\rangle \in S_{\alpha,\alpha'} \text{ and } z \subseteq s_{\alpha,\alpha'}\}$. This implies that $f(S(y))=\{\langle sid,s_{\alpha,\alpha'}\rangle \mid \langle sid,s\rangle \in S_{\alpha,\alpha'} \text{ and } z \subseteq s_{\alpha,\alpha'}\}$. We have $|S(y)|=|f(S(y))|$, as function f is bijective. Therefore $support(S,y)=|S(y)|=|f(S(y))|=support(S^{\alpha,\alpha'},z)$. Hence the lemma.

Lemma 2 (*What does set $\alpha.patterns(S^{\alpha,\alpha'}).\alpha'$ denote for patterns(S) ?*). The complete set of sequential patterns of S which are prefixed with α and suffixed with α' is equal to $\alpha.patterns(S^{\alpha,\alpha'}).\alpha'$, where function patterns denotes the complete set of sequential patterns of its unique argument.

Proof. Let x be a sequence. Assume that $x \in \alpha.patterns(S^{\alpha,\alpha'}).\alpha'$. This means that $x=\alpha.z.\alpha'$ for some $z \in patterns(S^{\alpha,\alpha'})$. From lemma 1, we have $support(S^{\alpha,\alpha'},z) = support(S,\alpha.z.\alpha')$. It comes that, x is also a sequential pattern in S as z is a sequential pattern in $S^{\alpha,\alpha'}$. Thus, $\alpha.patterns(S^{\alpha,\alpha'}).\alpha'$ is included in the set of sequential patterns of S which are prefixed with α and suffixed with α'.

Now, assume that x is a sequential pattern of S which is prefixed with α and suffixed with α'. We have $x=\alpha.z.\alpha'$ for some sequence z. From lemma 1, we have $support(S^{\alpha,\alpha'},z)=support(S, \alpha.z.\alpha')$. It comes that, z is also a sequential pattern in $S^{\alpha,\alpha'}$ as x is a sequential pattern in S. This means that $z \in patterns(S^{\alpha,\alpha'})$. Thus, the complete set of sequential patterns of S which are prefixed with α and suffixed with $\alpha'$is included in $\alpha.patterns(S^{\alpha,\alpha'}). \alpha'$.

Hence the lemma.

Lemma 3 (*Sequence decomposition lemma*). Let $\beta=\langle e'_1e'_2 \ldots e'_m\rangle$ be a sequence such that $\beta=\gamma.\mu$ for some non-empty prefix γ and some non-empty suffix μ. Either $\gamma=\langle e'_1 \ldots e'_k\rangle$ and $\mu=\langle e'_{k+1} \ldots e'_m\rangle$ for some integer k or $\gamma=\langle e'_1\ldots e'_{k-1}\gamma_{k}\_\rangle$, $\mu=\langle\_\mu_k e'_{k+1} \ldots e'_m\rangle$, $e'_k=\gamma_{k}\_ \cup \_\mu_k$, all the items in $\gamma_{k}\_$ are alphabetically before those in $\_\mu_k$ (this implies that $\gamma_{k}\_ \cap \_\mu_k=\emptyset$), $\gamma_{k}\_ \neq\emptyset$ and $\mu_{k}\_ \neq\emptyset$ for some integer k such that $1\leq k \leq m$.

Proof. Let $\beta=\langle e'_1e'_2 \ldots e'_m\rangle = \gamma.\mu$, where $\gamma\neq\varepsilon$ and $\mu\neq\varepsilon$. According to definitions 10 and 11, $\gamma=\langle e'_1 \ldots e'_{k-1}\gamma_{k}\_\rangle$, $\mu=\langle\_\mu_k e'_{k+1} \ldots e'_m\rangle$, $e'_k= \gamma_{k}\_ \cup \_\mu_k$ and all the items in $\gamma_{k}\_$ are alphabetically before those in $\_\mu_k$ for some integer k ($1\leq k \leq m$). We have the following cases:

Case 1: k = 1. This means that $\gamma=\langle\gamma_{1}\_\rangle$ and $\mu=\langle\_\mu_1 e'_2 \ldots e'_m\rangle$. We have $\gamma_{1}\_ \neq\emptyset$ as $\gamma\neq\varepsilon$. We also have $\_\mu_1\neq e'_1$ as the contrary, i.e. $\_\mu_1 = e'_1$, implies that $\gamma= \varepsilon$. If $\_\mu_1 = \emptyset$, $\gamma_{1}\_=e'_1$ and it comes that $\gamma=\langle e'_1\rangle$ and $\mu=\langle e'_2 \ldots e'_m\rangle$, which corresponds to the first half of the claim of the lemma. Otherwise, we have $\gamma_{1}\_ \neq\emptyset$ and $\mu_{1}\_ \neq\emptyset$, which leads to the second half of the claim of the lemma.

Case 2: k=m. This means that $\gamma=\langle e'_1 \ldots e'_{m-1}\gamma_{m}\_\rangle$ and $\mu=\langle\_\mu_m\rangle$. We have $\_\mu_m\neq\emptyset$ as $\mu\neq\varepsilon$. We also have $\gamma_{m}\_ \neq e'_m$ as the contrary, i.e. $\gamma_{m}\_ = e'_m$, implies that $\mu= \varepsilon$. If $\gamma_{m}\_ = \emptyset$, $\_\mu_m=e'_m$ and it comes that $\gamma=\langle e'_1 \ldots e'_{m-1}\rangle$ and $\mu=\langle e'_m\rangle$, which corresponds to the first half of the claim of the lemma. Otherwise, we have $\gamma_{m}\_ \neq\emptyset$ and $\mu_m\neq\emptyset$, which leads to the second half of the claim of the lemma.

Case 3: k≠1, k≠m and $\gamma_{k}\_=\emptyset$. This implies that $\mu_{k}\_=e'_k$. It comes that $\gamma=\langle e'_1 \ldots e'_{k-1}\rangle$ and $\mu=\langle e'_k \ldots e'_m\rangle$, which corresponds to the first half of the claim of the lemma.

Case 4: k≠1, k≠m and $\_\mu_k=\emptyset$. This case is similar to case 3. We have $\gamma_{k}\_=e'_k$. This implies that $\gamma=\langle e'_1 \ldots e'_k\rangle$ and $\mu=\langle e'_{k+1} \ldots e'_m\rangle$, which corresponds to the first half of the claim of the lemma.

Case 5: k≠1, k ≠ m, $\gamma_{k}\_ \neq\emptyset$ and $\_\mu_k\neq\emptyset$. This leads to the second half of the claim of the lemma. □

Definition 14 (*Static and dynamic search-space partitioning*). A search space partition is said to be static iff it is fully specified before the beginning of the mining process. It is said to be dynamic iff it is constructed during the mining process.

Lemma 4 (*Search-space partitioning based on prefix and/or suffix*)**.** We have the following.

1. Let $\{x_1, x_2, \ldots, x_n\}$ be the complete set of length-1 sequential patterns in a sequence database S. The complete set of sequential patterns in S can be divided into n disjoint subsets in two different ways:
    a. *Prefix-item-based search-space partitioning* (Pei et al., 2004): The i-th subset ($1\leq i \leq n$) is the set of sequential patterns with prefix x_i.
    b. *Suffix-item-based search-space partitioning* (Pei et al., 2004): The i-th subset ($1\leq i \leq n$) is the set of sequential patterns with suffix x_i.
2. Let α be a length-l sequential pattern and $\{\beta_1, \beta_2, \ldots, \beta_p\}$ be the set of all length-(l+1) sequential patterns with prefix α. Let α' be a length-l′ sequential pattern and $\{\gamma_1, \gamma_2, \ldots, \gamma_q\}$ be the set of all length-(l′+1) sequential patterns with suffix α'. We have:
    a. *Prefix-based search-space partitioning* (Pei et al., 2004): The complete set of sequential patterns with prefix α, except for α itself, can be divided into p disjoint subsets. The i-th subset ($1\leq i\leq p$) is the set of sequential patterns prefixed with β_i.
    b. *Suffix-based search-space partitioning* (Pei et al., 2004): The complete set of sequential patterns with suffix α', except for α' itself, can be divided into q disjoint subsets. The j-th subset ($1\leq j\leq q$) is the set of sequential patterns suffixed with γ_j.
    c. *Prefix-suffix-based search-space partitioning*: The complete set of sequential patterns with prefix α and suffix α', and of length greater or equal to l+l′+1, can be divided into p or q disjoint subsets. In the first partition, the i-th subset ($1\leq i \leq p$) is the set of sequential patterns prefixed with β_i and suffixed with α'. In the second partition, the j-th subset ($1\leq j\leq q$) is the set of sequential patterns prefixed with α and suffixed with γ_j.

Proof. Parts (1.a) and (2.a) of the lemma are proven in (Pei et al., 2004). The proof of parts (1.b) and (2.b) of the lemma is similar to the proof of parts (1.a) and (2.a). Thus, we only show the correctness of part (2.c).

Let μ be a sequential pattern of length greater or equal to l+l′+1, with prefix α and with suffix α', where α is of length l and α' is of length l′. The length-(l+1) prefix of μ is a sequential pattern according to an Apriori principle which states that a subsequence of a sequential pattern is also a sequential pattern. Furthermore, α is a prefix of the length-(l+1) prefix of μ, according to the definition of prefix. This implies that there exists some i ($1\leq i\leq p$)

such that β_i is the length-(l+1) prefix of μ. Thus μ is in the i-th subset of the first partition. On the other hand, since the length-k prefix of a sequence is unique, the subsets are disjoint and this implies that μ belongs to only one determined subset. Thus, we have (2.c) for the first partition. The proof of (2.c) for the second partition is similar. Therefore we have the lemma. □

Corollary 2 (*Partitioning S with sets x_i.patterns($S^{xi,\varepsilon}$) and patterns($S^{\varepsilon,xi}$).x_i*). Let $\{x_1, x_2, \dots, x_n\}$ be the complete set of length-1 sequential patterns in a sequence database S. The complete set of sequential patterns in S can be divided into n disjoint subsets in two different ways:

1. *Prefix-item-based search-space partitioning*: The i-th subset ($1\le i \le n$) is x_i.patterns($S^{xi,\varepsilon}$), where function patterns denotes the set of sequential patterns of its unique argument.
2. *Suffix-item-based search-space partitioning*: The i-th subset ($1\le i\le n$) is patterns($S^{\varepsilon,xi}$).x_i.

Proof. According to part 1.(a) of lemma 4, the i-th subset is the set of sequential patterns which are prefixed with x_i. From lemma 2, this subset is x_i.patterns($S^{xi,\varepsilon}$). Similarly, according to part 1.(b) of lemma 4, the i-th subset is the set of sequential patterns suffixed with x_i. From lemma 2, this subset is patterns($S^{\varepsilon,xi}$).x_i. □

Lemma 5 (*A linear ordering induces a recursive pruning and partitioning*). A linear ordering induces a recursive pruning and partitioning of the search space. The recursive partitioning is static if the linear ordering is static and dynamic otherwise.

Proof. Let us consider the initial sequence database S, two integer numbers l and l′, a length-l sequential pattern α, a length-l′ sequential pattern α', and a linear ordering L_0=o_0-o_1-o_2 … o_{n-1}-growth. Note that $\varepsilon.S^{\varepsilon,\varepsilon}.\varepsilon$=S is the starting database of the recursive pruning and partitioning of the search space. In the following, we show how L_0 induces a recursive pruning and partitioning of $\alpha.S^{\alpha,\alpha'}.\alpha'$.

Case 1: $o_0 \in$ {prefix}. Let $\{\beta_1.\alpha', \beta_2.\alpha', \dots, \beta_p.\alpha'\}$ be the set of all length-(l+l′+1) sequential patterns with respect to database $\alpha.S^{\alpha,\alpha'}.\alpha'$, prefixed with α and suffixed with α'. From lemma 3, either $\beta_I=\alpha.\langle(x_i)\rangle$ or $\beta_I=\alpha.\langle(\_x_i)\rangle$, where x_i is an item and $1\le i\le p$. This implies that $X=\{\langle x_1\rangle, \langle x_2\rangle, \dots, \langle x_p\rangle\}$ is the complete set of length-1 sequential patterns with respect to database $S^{\alpha,\alpha'}$. It comes that any item that does not belong to X is not frequent with respect to $S^{\alpha,\alpha'}$. Thus, any sequence that contains an item that does not belong to X is not frequent with respect to $S^{\alpha,\alpha'}$ according to an Apriori principle which states that any supersequence of an infrequent sequence is also infrequent. Because of this, all the infrequent items with respect to $S^{\alpha,\alpha'}$ are removed from the z part (also called the middle part) of all sequence $\alpha.z.\alpha'\in\alpha.S^{\alpha,\alpha'}.\alpha'$. This pruning step leads to a new sequence database $\alpha.S'^{\alpha,\alpha'}.\alpha'$ whose middle parts of sequences do not contain infrequent items with respect to $S^{\alpha,\alpha'}$. Then, $\alpha.S'^{\alpha,\alpha'}.\alpha'$ is partitioned according to part (2.c) of lemma 4. The i-th sub-database ($1\le i\le p$) of $\alpha.S'^{\alpha,\alpha'}.\alpha'$, denoted $\alpha.x_i.S'^{\alpha.xi,\alpha'}.\alpha'$, is the set of subsequences of $\alpha.S'^{\alpha,\alpha'}.\alpha'$ with prefix $\beta_i=\alpha.x_i$ and with suffix α'. Each sub-database is in turn recursively pruned and partitioned according to L_1=o_1-o_2… o_{n-1}-growth linear ordering.

Case 2: $o_0\in$ {suffix}. Let $\{\alpha.\gamma_1, \alpha.\gamma_2, \dots, \alpha.\gamma_p\}$ be the set of all length-(l+l′+1) sequential patterns with respect to database $\alpha.S^{\alpha,\alpha'}.\alpha'$, prefixed with α and suffixed with α'. From lemma 3, either $\gamma_I=\langle(x_i)\rangle.\alpha'$ or $\gamma_I=\langle(x_i\_)\rangle.\alpha'$ ($1\le i\le p$). As in case 1, $\alpha.S'^{\alpha,\alpha'}.\alpha'$ is partitioned according to part (2.c) of lemma 4. The i-th sub-database ($1\le i\le p$) of $\alpha.S'^{\alpha,\alpha'}.\alpha'$, denoted $\alpha.S'^{\alpha,xi.\alpha'}.x_i.\alpha'$,is the set of subsequences of $\alpha.S'^{\alpha,\alpha'}.\alpha'$ with prefix α and with suffix $\gamma_I=x_i.\alpha'$. As in case 1, each sub-database is in turn recursively pruned and partitioned according to L_1=o_1-o_2 … o_{n-1}-growth linear ordering.

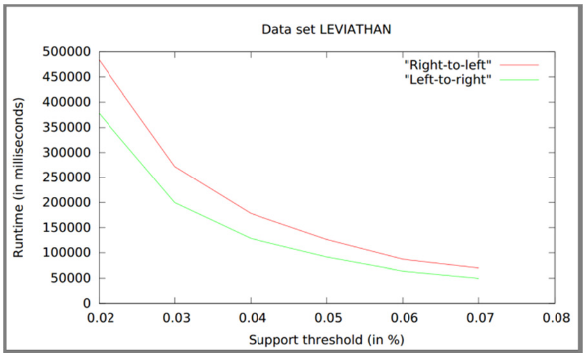
Case 3: $o_0\in$ {*}. A pattern-growth direction is determined during the mining process. Then, $\alpha.S^{\alpha,\alpha'}.\alpha'$ is recursively pruned and partitioned as in case 1 if the determined direction is left-to-right and as in case 2 otherwise. From definitions 6 and 14 it is easy to see that the recursive partitioning is static if the linear ordering is static and dynamic otherwise. □

5. Experimental results

The data set used here is collected from the webpage of SPMF software (Fournier-Viger et al., 2014). This webpage (http://www.philippe-fournier-viger.com/spmf/index.php) provides large data sets in SPMF format that are often used in the data mining literature for evaluating and comparing algorithm performance.

Experiments were performed on real-life data sets. The first data set is *LEVIATHAN*. It contains 5834 sequences and 9025 distinct items. The second data set is *Kosarak*. It is a very large data set containing 990000 sequences of click-stream data from an hungarian news portal. The third data set is *BIBLE*. It is a conversion of the Bible into a sequence database (each word is an item). It contains 36 369 sequences and 13905 distinct items. The fourth data set is *BMSWebView2* (*Gazelle*). It is called here *BMS2*. It contains 59601 sequences of clickstream data from e-commerce and 3340 distinct items.

All experiments were done on a 4-cores of 2.16GHz Intel(R) Pentium(R) CPU N3530 with 4 gigabytes main memory, running Ubuntu 14.04 LTS. The algorithms are implemented in Java and grounded on SPMF software (Fournier-Viger et al., 2014). The experiments consisted of running the pattern-growth algorithms related to the left-to-right and the right-to-left orderings on each data set while decreasing the support threshold until algorithms became too long to execute or ran out of memory. The performances are presented in figures 1, 2, 3 and 4. These figures show that the order in which patterns grow has a significant influence on the performances.



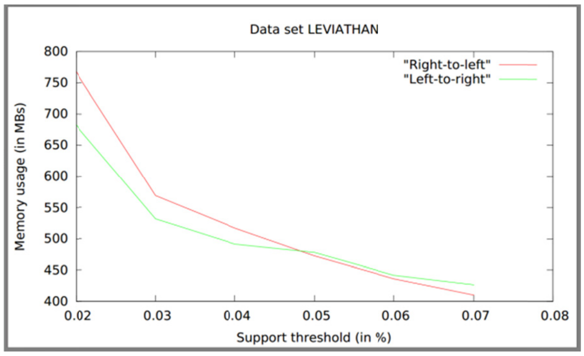
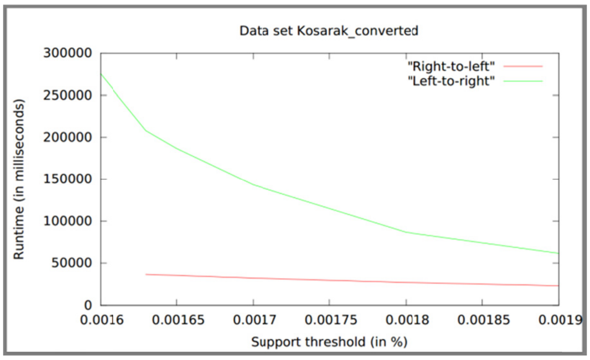


Figure 1. Performances of left-to-right and right-to-left pattern-growth orderings on the real-life data set LEVIATHAN. The left-to-right pattern-growth ordering is 1.27-1.4 times faster, and requires less memory if the support threshold is less than 0.05 and a little more memory otherwise



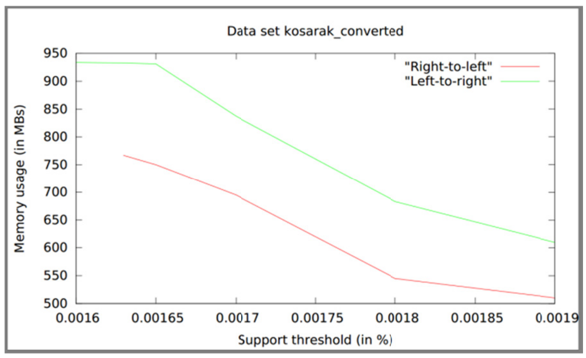
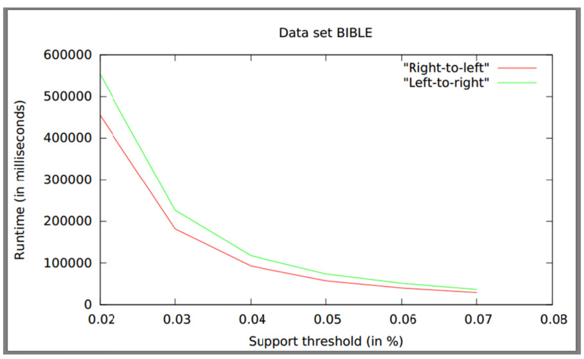


Figure 2. Performances of left-to-right and right-to-left pattern-growth orderings on the real-life data set kosarak_converted. The right-to-left pattern-growth ordering is 2.6-5.6 times faster and requires almost 1.2 times less memory than the other direction



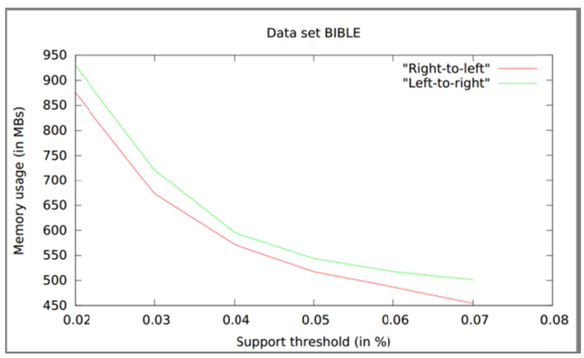


Figure 3. Performances of left-to-right and right-to-left pattern-growth orderings on the real-life data set BIBLE. The right-to-left pattern-growth ordering is 1.21-1.25 times faster and requires almost 1.04-1.10 times less memory than the other ordering

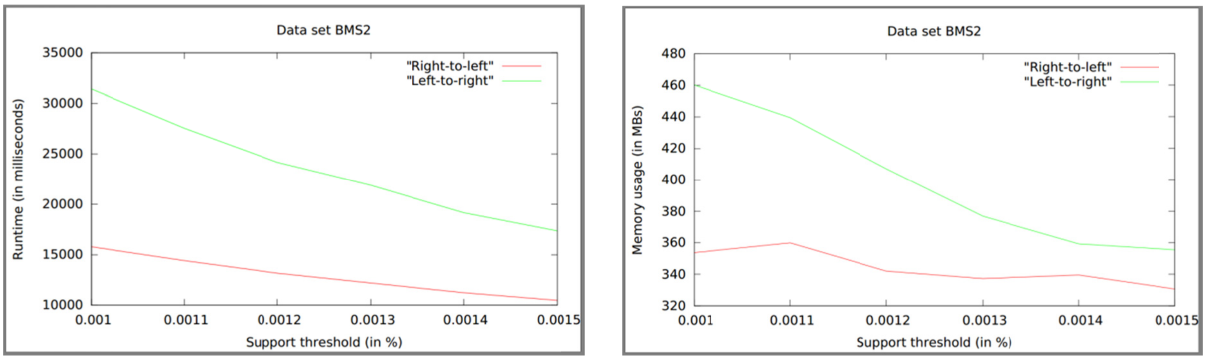


Figure 4. Performances of left-to-right and right-to-left pattern-growth orderings on the real-life data set BMS2. The right-to-left pattern-growth ordering is 1.5-2 times faster and requires almost 1.07-1.3 times less memory than the other ordering

6. Conclusion

In this article, we have study the theoretical foundations of pattern growth-based sequential pattern mining algorithms. The important key concepts of the pattern-growth approach are revisited, formally defined and extended. A new class of pattern-growth algorithms inspired from a new class of pattern-growth orderings, called linear orderings, is introduced. Issues of this new class of pattern-growth algorithms related to search space pruning and partitioning are investigated. Experimentations show that the order in which patterns grow has a significant influence on the performances.

References

Agrawal, R., Imielinski, T., & Swami, A. N. (1993). Mining association rules between sets of items in large databases, 207–216.

Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules in large databases. In VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile, 487–499.

Agrawal, R., & Srikant, R. (1995). Mining sequential patterns. In Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan, 3–14.

Ayres, J., Flannick, J., Gehrke, J., & Yiu, T. (2002). Sequential pattern mining using a bitmap representation. In Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, July 23-26, 2002, Edmonton, Alberta, Canada, 429–435.

Dam, T., Li, K., Fournier-Viger, P., & Duong, Q. (2016). An efficient algorithm for mining top-rank-k frequent patterns. *Appl. Intell., 45*(1), 96–111.

El-Sayed, M., Ruiz, C., & Rundensteiner, E. A. (2004). Fs-miner: efficient and incremental mining of frequent sequence patterns in web logs. In Sixth ACM CIKM International Workshop on Web Information and Data Management (WIDM 2004), Washington, DC, USA, 12-13, 128–135.

Fournier-Viger, P., Gomariz, A., Gueniche, T., Soltani, A., Wu, C., & Tseng, V. S. (2014). SPMF: a java open-source pattern mining library. *Journal of Machine Learning Research, 15*(1), 3389–3393.

Garofalakis, M. N., Rastogi, R., & Shim, K. (1999). SPIRIT: sequential pattern mining with regular expression constraints. In VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK, 223–234.

Gouda, K., Hassaan, M., & Zaki, M. J. (2007). Prism: A primal-encoding approach for frequent sequence mining. In Proceedings of the 7th IEEE International Conference on Data Mining (ICDM 2007), October 28-31, 2007, Omaha, Nebraska, USA, 487–492.

Gouda, K., Hassaan, M., & Zaki, M. J. (2010). Prism: An effective approach for frequent sequence mining via prime-block encoding. J. Comput. Syst. Sci., 76(1):88–102.

Han, J., Pei, J., Mortazavi-Asl, B., Chen, Q., Dayal, U., & Hsu, M. (2000). Freespan: frequent pattern-projected sequentialpattern mining. In Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, Boston, MA, USA, August 20-23, 355–359.

Hsieh, C., Yang, D., & Wu, J. (2008). An efficient sequential pattern mining algorithm based on the 2-sequence matrix. In Workshops Proceedings of the 8th IEEE International Conference on Data Mining (ICDM 2008), December 15-19, 2008, Pisa, Italy, 583–591.

Lin, J. C., Gan, W., Fournier-Viger, P., Hong, T., &Tseng, V. S. (2016). Efficient algorithms for mining high-utility itemsets in uncertain databases. *Knowl. Based Syst., 96*, 171–187.

Lin, J. C., Gan, W., Fournier-Viger, P., Hong, T., & Tseng, V. S. (2016). Fast algorithms for mining high-utility itemsets with various discount strategies. *Advanced Engineering Informatics, 30*(2), 109–126.

Lin, J. C., Gan, W., Fournier-Viger, P., Hong, T., & Zhan, J. (2016). Efficient mining of high-utility itemsets using multiple minimum utility thresholds. Knowl.-Based Syst., 113, 100–115.

Lin, J. C., Li, T., Fournier-Viger, P., Hong, T., Zhan, J., & Voznak, M. (2016). An efficient algorithm to mine high average-utility itemsets. *Advanced Engineering Informatics, 30*(2), 233–243.

Mabroukeh, N. R., &Ezeife, C. I. (2010). A taxonomy of sequential pattern mining algorithms. *ACM Comput. Surv., 43*(1), 3.

Masseglia, F., Cathala, F., & Poncelet, P. (1998). *The PSP approach for mining sequential patterns.* In Principles of Data Mining and Knowledge Discovery, Second European Symposium, PKDD '98, Nantes, France, September 23-26, Proceedings, 176–184.

Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U., & Hsu, M. (2001). *Prefixspan: Mining sequential patterns by prefix-projected growth.* In Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany, 215–224.

Pei, J., Han, J., Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Dayal, U., & Hsum, M. (2004). Mining sequential patterns by pattern-growth: The prefixspan approach. IEEE Trans. *Knowl. Data Eng., 16*(11), 1424–1440.

Pei, J., Han, J., Mortazavi-Asl, B., & Zhu, H. (2000). *Mining access patterns efficiently from web logs.* In Knowledge Discovery and Data Mining, Current Issues and New Applications, 4th Pacific-Asia Conference, PADKK 2000, Kyoto, Japan, April 18-20, Proceedings, 396–407.

Savary, L., & Zeitouni, K. (2005). *Indexed bit map (IBM) for mining frequent sequences.* In Knowledge Discovery in Databases: PKDD 2005, 9th European Conference on Principles and Practice of Knowledge Discovery in Databases, Porto, Portugal, October 3-7, 2005, Proceedings, 659–666.

Seno, M., & Karypis, G.. (2002). *Slpminer: An algorithm for finding frequent sequential patterns using length-decreasing support constraint.* In Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002), 9-12 December 2002, Maebashi City, Japan, 418–425.

Yang, Z., & Kitsuregawa, M. (2005). LAPIN-SPAM: an improved algorithm for mining sequential pattern. In Proceedings of the 21st International Conference on Data Engineering Workshops, ICDE 2005, 5-8 April 2005, Tokyo, Japan, 1222.

Yang, Z., Wang, Y., & Kitsuregawa, M. (2007). LAPIN: Effective sequential pattern mining algorithms by last position induction for dense databases. In Advances in Databases: Concepts, Systems and Applications, 12th International Conference on Database Systems for Advanced Applications, DASFAA 2007, Bangkok, Thailand, April 9-12, 2007, Proceedings, 1020–1023.

Zaki, M. J. (2000). Sequence mining in categorical domains: Incorporating constraints. In Proceedings of the 2000 ACM CIKM International Conference on Information and Knowledge Management, McLean, VA, USA, November 6-11, 2000, 422–429.

Zaki, M. J. (2001). SPADE: an efficient algorithm for mining frequent sequences. *Machine Learning, 42*(1/2), 31–60.