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

Kenmogne Edith Belise

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; Lin et 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 \( I = \{i_1, i_2, \ldots, i_n\} \) 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 \( \langle s_1, s_2, \ldots, s_p \rangle \), 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 \( \alpha \) is denoted \( |\alpha| \). A sequence \( \alpha = (a_1, a_2, \ldots, a_n) \), is called subsequence of another sequence \( \beta = (b_1, b_2, \ldots, b_m) \) and \( \beta \) is a supersequence of \( \alpha \), denoted as \( \alpha \subseteq \beta \), if there exist integers \( 1 \leq j_1 < j_2 < \ldots < j_m \leq \alpha \) such that \( a_1 \subseteq b_{j_1}, a_2 \subseteq b_{j_2}, \ldots, a_n \subseteq b_{j_n} \). Symbol \( \epsilon \) denotes the empty sequence.

We are given a database \( S \) of input-sequences. A sequence database is a set of tuples of the form \( \langle \text{sid}, s \rangle \) where \( \text{sid} \) is a sequence_id and \( s \) a sequence. A tuple \( \langle \text{sid}, s \rangle \) is said to contain a sequence \( \alpha \) if \( \alpha \) is a subsequence of \( s \). The support of a sequence \( \alpha \) in a sequence database \( S \) is the number of tuples in the database containing \( \alpha \), i.e. support(\( S, \alpha \)) = |\{ \langle \text{sid}, s \rangle \mid \langle \text{sid}, s \rangle \in S \text{ and } \alpha \subseteq s \}|.

It can be denoted as support(\( \alpha \)) 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 \( \alpha \) is called a sequential pattern in the sequence database \( S \) if support(\( S, \alpha \)) \geq \text{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.
There are two pattern-growth directions, namely left-to-right and right-to-left directions. 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 \( \ast \)-growth, prefix-growth and suffix-growth orderings, and denoted \( o_0-o_1-o_2 \ldots o_{n-1}\)-growth for some \( n \), where \( o_i \in \{ \text{prefix, suffix, } \ast \} \) \( (0 \leq i \leq n-1) \). It is said to be static iff \( o_i \in \{ \text{prefix, suffix} \} \) for all \( i \in \{ 0, 1, 2, \ldots, n-1 \} \). Otherwise, it is said to be dynamic.

The \( o_0-o_1-o_2 \ldots o_{n-1}\)-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 \( i \) \( (0 \leq i \leq n-1) \) a prefix (resp. suffix) of that pattern if \( o_i \) denotes prefix (resp. suffix). If \( o_i \in \{ \ast \} \), 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 \( k \) \( (k \geq 4) \)**: Repeat step \( k \mod 4 \).

The prefix-suffix-\( \ast \)-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 \( x = (x_1x_2 \ldots x_m) \), another itemset \( x' = (x'_1x'_2 \ldots x'_m) \) \( (m \leq n) \) is called a prefix of \( x \) if and only if \( x'_i = x_i \) for all \( i \leq m \). If \( m < n \), the prefix is also denoted as \( x = (x_1x_2\ldots x_m) \).

**Definition 8** (The corresponding suffix of a prefix of an itemset). Let \( x = (x_1x_2 \ldots x_n) \) be a itemset. Let \( x' = (x_1x_2 \ldots x_m) \) \( (m \leq n) \) be a prefix of \( x \). Itemset \( x'' = (x_{m+1}x_{m+2} \ldots x_n) \) 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 \( 1 \leq m < n \), the suffix is also denoted as \( (x_{m+1}x_{m+2} \ldots x_n) \).

For example, for the itemset \( \text{iset} = (\text{abcd}_{-})(\text{efgh}) \) is the suffix with regards to the prefix \( (\text{abcd}_{-}) \), \( \text{iset} = (\text{abcd}_{-})(\text{efgh}) \), \( (\text{abcd}_{-}) \) is the prefix with regards to suffix \( (\text{gh}) \) and \( \text{iset} = (\text{abcd}_{-})(\text{gh}) \).

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

**Definition 9 ("\而是 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 \(\gamma=(e_1, e_2, \ldots, e_m, a)\) and \(\mu=(b'e_2, \ldots, e'_m)\) be two sequences, where e_i and e'_i are itemsets that do not contain the underscore symbol, a \(\in \{e_1, (_\text{items in e}), (\text{items in e'})\}\) and b \(\in \{e', (_\text{items in e'}), (\text{items in e' '}), (\text{items in e''})\}\). The dot operator is defined as follows.

1. \(e \cdot e' = ee'\)
2. \(e \cdot (_\text{items in e'}) = (\text{items in e} \cup \text{e'})\)
3. \(e \cdot (\text{items in e'}) = e (\text{items in e'})\)
4. \(e \cdot (_\text{items in e'}) = (\text{items in e} \cup \text{e'})\)
5. \(\text{items in e} \cdot e' = (\text{items in e} \cup \text{e'})\)
6. \((\text{items in e}) \cdot (\text{items in e'}) = (\text{items in e} \cup \text{e'})\)
7. \((\text{items in e}) \cdot (\text{items in e'}) = (\text{items in e} \cup \text{e'})\)
8. \((\text{items in e}) \cdot (\text{items in e'}) = (\text{items in e} \cup \text{e'})\)
9. \((\text{items in e}) \cdot e' = (\text{items in e})\)
10. \((\text{items in e}) \cdot (\text{items in e'}) = (\text{items in e})\)
11. \((\text{items in e}) \cdot (\text{items in e'}) = (\text{items in e})\)
12. \((\text{items in e}) \cdot (\text{items in e'}) = (\text{items in e})\)
13. \((\text{items in e}) \cdot e' = (\text{items in e})\)
14. \((\text{items in e}) \cdot (\text{items in e'}) = (\text{items in e})\)
15. \((\text{items in e}) \cdot (\text{items in e'}) = (\text{items in e})\)
16. \((\text{items in e}) \cdot (\text{items in e'}) = (\text{items in e})\)
17. \(\gamma \mu = (\alpha_1, \ldots, \alpha_n, a, b'e_2, \ldots, e'_m)\)

For example, \(s=(a(ab)(ac)(efgh))=(a)((a)_.)((b.)_.)((c.)_.)((e)_.)((f.)_.)((g.)_.)((h.)_.)\) and \(s'=((a)_.)((a)_.)((b.)_.)((c.)_.)((e)_.)((f.)_.)((g.)_.)((h.)_.)\).

**Definition 10 (Prefix of a sequence)** (Pei et al., 2004). Suppose all the items within an element are listed alphabetically. Given a sequence \(\alpha=(e_1, e_2, \ldots, e_n)\), a sequence \(\beta=(e'_1, e'_2, \ldots, e'_m)\) (m \(\leq\) n) is called a prefix of \(\alpha\) if and only if 1) \(e'_1 = e_i\) for all \(i \leq m\); 2) \(e'_m \subseteq e_n\); and 3) all the frequent items in \(e_n - e'_m\) are alphabetically sorted after those in \(e'_m\). If \(e'_m \neq \emptyset\) and \(e'_m \subseteq e_n\) the prefix is also denoted as \((\alpha_1, \alpha_2, \ldots, \alpha_m)\) (items in \(e'_m\)).

**Definition 11 (The corresponding suffix of a prefix of a sequence)** (Pei et al., 2004). Given a sequence \(\alpha=(e_1, e_2, \ldots, e_n)\). Let \(\beta=(e_1, e_2, \ldots, e_m)\) (m \(\leq\) n) be a prefix of \(\alpha\). Sequence \(\gamma=(e'_m, e_{m+1}, \ldots, e_n)\) is called the suffix of \(\alpha\) with regards to prefix \(\beta\), denoted as \(\gamma = \alpha/\beta\), where \(e'_m = e_n - e'_m\). We also denote \(\alpha = \beta.\gamma\). Note, if \(\beta = \alpha\), the suffix of \(\alpha\) with regards to \(\beta\) is empty. If \(e'_m\) is not empty, the suffix is also denoted as \((\_\text{items in e})_m \cup \_\text{e} \cup \_\text{e'}\).

For example, for the sequence \(s=(a(ab)(ac)(efgh))\), \(\langle a\rangle\) is the suffix with regards to the prefix \(\langle a\rangle\), \(\langle bc\rangle\) is the suffix with regards to the prefix \(\langle a\rangle\), \(\langle ac\rangle\) is the suffix with regards to the prefix \(\langle ab\rangle\), and \(\langle ab\rangle\) is the prefix with regards to the suffix \(\langle c\rangle\).

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

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

For example, consider sequence \(s=(a(ab)(ac)(efgh))\), we have \(spc(s, (a))=(a), spc(s, (ab))=(a(ab)), spc(s, (ac))=(a(ab)), ssc(s, (c))=(c)(efgh), ssc(s, (bc))=(bc)(efgh), sssc(s, (ab))=(ab)(c)(efgh), sssc(s, (a))=(ac)(efgh), spc(s, ((ab)(a))) 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 $\alpha$ and suffix $\alpha'$, we define two sets of the sequence database $S$ as follows. We denote $S_{\alpha\alpha'}$ the set of subsequences of $S$ prefixed with $\alpha$ and suffixed with $\alpha'$ which are obtained by replacing the left and right parts of canonical decompositions respectively with $\alpha$ and $\alpha'$. We have $S_{\alpha\alpha'}=\{(sid,\alpha,y_{\alpha\alpha'},\alpha')|(sid,y)\in S$ and $y=spc(y,\alpha)y_{\alpha\alpha'},ssc(y,\alpha')\}$. We denote $S^{\alpha\epsilon}$ the set of subsequences which are obtained by removing the left and right parts of canonical decompositions. We have $S^{\alpha\epsilon}=\{(sid,y_{\alpha\alpha'})|(sid,y)\in S$ and $y=spc(y,\alpha)y_{\alpha\alpha'},ssc(y,\alpha')\}$. We also have $S=S_{\epsilon\epsilon}$ and $S=S^{\epsilon\epsilon}$ as $\epsilon$ denotes the empty sequence.

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

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

1. $(\alpha.\alpha').\alpha''=(\alpha.\alpha').\alpha''$
2. $\alpha.(\alpha'.S)=(\alpha.\alpha').S$
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 $\alpha$ and $\alpha'$, for any sequence $y$ prefixed with $\alpha$ and suffixed with $\alpha'$, i.e. $y=\alpha.z.\alpha'$ for some sequence $z$, we have $support(S.y)=support(S^{\alpha\alpha'}.y)$.

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

Let's prove that function $f$ is injective. Consider two tuples of $S$, $(sid, y)$ and $(sid', y')$, each having a canonical decomposition following prefix $\alpha$ and suffix $\alpha'$. Assume that $f(sid, spc(y,\alpha)y_{\alpha\alpha'},ssc(y,\alpha'))=f(sid', spc(y',\alpha)y_{\alpha\alpha'},ssc(y',\alpha'))$. This implies that $(sid,y_{\alpha\alpha'})=(sid',y_{\alpha\alpha'})$, which in turn implies that $sid=sid'$. This implies that tuple $(sid,y)$ is equal to $sid'$ as the identifier of any tuple is unique. It comes that $y=y'$. Thus $(sid, spc(y,\alpha)y_{\alpha\alpha'},ssc(y,\alpha'))=(sid', spc(y',\alpha)y_{\alpha\alpha'},ssc(y',\alpha'))$. Therefore function $f$ is injective.

Let's prove that function $f$ is surjective. Consider $(sid, z_{\alpha\alpha'})\in S^{\alpha\alpha'}$, where $(sid,z)$ belongs to $S$ and admits a canonical decomposition following prefix $\alpha$ and suffix $\alpha'$. From the definition of function $f$, $f(sid, spc(z,\alpha)z_{\alpha\alpha'},ssc(z,\alpha'))=(sid,z_{\alpha\alpha'})$. This means that $(sid,z_{\alpha\alpha'})\in S^{\alpha\alpha'}$ 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 $\alpha$ and suffixed with $\alpha'$, i.e. $y=\alpha.z.\alpha'$ for some sequence $z$. Denote $S(y)=\{(sid,s)\mid (sid,s)\in S$ 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 $\alpha$ and suffix $\alpha'$ and containing sequence $z$ in their middle part. It comes that $S(y)=\{(sid,s)\mid (sid,s)\in S_{\alpha\alpha'}$ and $z\subseteq s_{\alpha\alpha'}\}$. This implies that $f(S(y))=\{|(sid,spc(z,\alpha)z_{\alpha\alpha'},ssc(z,\alpha'))\mid (sid,s)\in S_{\alpha\alpha'}$ 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 $\alpha.patterns(S^{\alpha\alpha'})$ denote for patterns(S)?)**. The complete set of sequential patterns of $S$ which are prefixed with $\alpha$ and suffixed with $\alpha'$ is equal to $\alpha.patterns(S^{\alpha\alpha'})$, where function patterns denotes the complete set of sequential patterns of its unique argument.

**Proof**. Let $x$ be a sequence. Assume that $x\in \alpha.patterns(S^{\alpha\alpha'})$. 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, $\alpha.patterns(S^{\alpha\alpha'})$ is included in the set of sequential patterns of $S$ which are prefixed with $\alpha$ and suffixed with $\alpha'$. Now, assume that $x$ is a sequential pattern of $S$ which is prefixed with $\alpha$ and suffixed with $\alpha'$. 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 $\alpha$ and suffixed with $\alpha'$ is included in $\alpha.patterns(S^{\alpha\alpha'})$. $\alpha'$. 
Hence the lemma.

**Lemma 3 (Sequence decomposition lemma).** Let $\beta=(e'_1 e'_2 \ldots e'_m)$ be a sequence such that $\beta=\gamma \mu$ for some non-empty prefix $\gamma$ and some non-empty suffix $\mu$. Either $\gamma=(e'_1 \ldots e'_k)$ and $\mu=(e'_{k+1} \ldots e'_m)$ for some integer $k$ or $\gamma=(e'_1 \ldots e'_k \gamma)$, $\mu=(\mu_k e'_k \ldots e'_m)$, $e'=\gamma \cup \mu_k$ and all the items in $\gamma$ 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 $\beta=(e'_1 e'_2 \ldots e'_m)=\gamma \mu$, where $\gamma \neq \mu$ and $\mu \neq \varepsilon$. According to definitions 10 and 11, $\gamma=(e'_1 \ldots e'_k \gamma)$, $\mu=(\mu_k e'_k \ldots e'_m)$, $e'=\gamma_k \cup \mu_k$ 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 $\gamma=(\gamma_1)$ and $\mu=(\mu_1 e'_2 \ldots e'_m)$. We have $\gamma_1 \neq \emptyset$ as the contrary, i.e. $\mu_1 \neq e'_1$, implies that $\gamma=\varepsilon$. If $\mu_1 = \emptyset$, $\gamma_1 = e'_1$ and it comes that $\gamma=(e'_1)$ and $\mu=(e'_2 \ldots e'_m)$, 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 $\gamma=(e'_1 \ldots e'_m)$ and $\mu=(\mu_m)$. We have $\gamma_m \neq \emptyset$ as the contrary, i.e. $\gamma_m = e'_m$ implies that $\mu=\varepsilon$. If $\gamma_m = \emptyset$, $\gamma_m = e'_m$ and it comes that $\gamma=(e'_1 \ldots e'_m)$ and $\mu=(\varepsilon)$, 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:** $k \neq 1, k \neq m$ and $\gamma_k = \emptyset$. This implies that $\mu_k = e'_k$. It comes that $\gamma=(e'_1 \ldots e'_k)$ and $\mu=(e'_k \ldots e'_m)$, which corresponds to the first half of the claim of the lemma.

**Case 4:** $k \neq 1, k \neq m$ and $\mu_k = \emptyset$. This case is similar to case 3. We have $\gamma_k = e'_k$. This implies that $\gamma=(e'_1 \ldots e'_k \gamma)$ and $\mu=(\varepsilon)$, which corresponds to the first half of the claim of the lemma.

**Case 5:** $k \neq 1, k \neq m$, $\gamma_k \neq \emptyset$ 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 $\alpha$ 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 $\alpha$. Let $\alpha'$ 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 $\alpha'$. We have:
   a. **Prefix-based search-space partitioning** (Pei et al., 2004): The complete set of sequential patterns with prefix $\alpha$, except for $\alpha$ 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 $\beta_i$.
   b. **Suffix-based search-space partitioning** (Pei et al., 2004): The complete set of sequential patterns with suffix $\alpha'$, except for $\alpha'$ 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 $\gamma_j$.
   c. **Prefix-suffix-based search-space partitioning**: The complete set of sequential patterns with prefix $\alpha$ and suffix $\alpha'$, and of length greater or equal to $l+l'+1$, can be divided into $p \times q$ disjoint subsets. In the first partition, the $i$-th subset ($1 \leq i \leq p$) is the set of sequential patterns prefixed with $\beta_i$ and suffixed with $\alpha'$. In the second partition, the $j$-th subset ($1 \leq j \leq q$) is the set of sequential patterns prefixed with $\alpha$ and suffixed with $\gamma_j$.

**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 $\mu$ be a sequential pattern of length greater or equal to $l+l'+1$, with prefix $\alpha$ and with suffix $\alpha'$, where $\alpha$ is of length $l$ and $\alpha'$ is of length $l'$. The length-$(l+1)$ prefix of $\alpha$ is a sequential pattern according to an Apriori principle which states that a subsequence of a sequential pattern is also a sequential pattern. Furthermore, $\alpha$ is a prefix of the length-$(l+1)$ prefix of $\mu$, according to the definition of the prefix. This implies that there exists some $i$ ($1 \leq i \leq p$)
such that $\beta_i$ is the length-(l+1) prefix of $\mu$. Thus $\mu$ 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 $\mu$ 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($S^{\leq \ell}$) and patterns($S^{\leq \ell}$).$x_i$).** Let $\{x_1, x_2, \ldots, x_n\}$ be the complete set of length-$l$ 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≤i≤n) is $x_i$.patterns($S^{\leq \ell}$), 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≤i≤n) is patterns($S^{\leq \ell}$).$x_i$.

**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($S^{\leq \ell}$). 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($S^{\leq \ell}$).$x_i$. □

**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 $\alpha$, a length-l’ sequential pattern $\alpha’$, and a linear ordering $L_0=\alpha_0-\alpha_1-\alpha_2 \ldots -\alpha_{n-1}$-growth. Note that $e.S^{\leq \ell}.e=S$ 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 $\alpha.S^{\leq \ell}.\alpha’$.

**Case 1:** $o_0 \in \{\text{prefix}\}$. Let $\{\beta_1, \alpha’_1, \beta_2, \alpha’_2, \ldots, \beta_p, \alpha’_p\}$ be the set of all length-(l+l’+1) sequential patterns with respect to database $\alpha.S^{\leq \ell}.\alpha’$, prefixed with $\alpha$ and suffixed with $\alpha’$. From lemma 3, either $\beta_1=\alpha.(x_i)$ or $\beta_1=\alpha.(x_j)$, where $x_i$ is an item and 1≤i≤p. This implies that $X=\{x_1, x_2, \ldots, x_n\}$ is the complete set of length-1 sequential patterns with respect to database $S^{\leq \ell}$. It comes that any item that does not belong to $X$ is not frequent with respect to $S^{\leq \ell}$. Thus, any sequence that contains an item that does not belong to $X$ is not frequent with respect to $S^{\leq \ell}$ 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 $S^{\leq \ell}$ are removed from the z part (also called the middle part) of all sequence $\alpha.z.\alpha’ \in \alpha.S^{\leq \ell}.\alpha’$. This pruning step leads to a new sequence database $\alpha.S^{\leq \ell}.\alpha’$ whose middle parts of sequences do not contain infrequent items with respect to $S^{\leq \ell}$. Then, $\alpha.S^{\leq \ell}.\alpha’$ is partitioned according to part (2.c) of lemma 4. The i-th sub-database (1≤i≤p) of $\alpha.S^{\leq \ell}.\alpha’$, denoted $\alpha.x_i.S^{\leq \ell}.x_i.\alpha’$, is the set of subsequences of $\alpha.S^{\leq \ell}.\alpha’$ with prefix $\beta_i=\alpha.x_i$ and with suffix $\alpha’$. Each sub-database is in turn recursively pruned and partitioned according to $L_1=\alpha_1-\alpha_2 \ldots -\alpha_{n-1}$-growth linear ordering.

**Case 2:** $o_0 \in \{\text{suffix}\}$. Let $\{\gamma_1, \gamma_2, \ldots, \gamma_p\}$ be the set of all length-(l+l’+1) sequential patterns with respect to database $\alpha.S^{\leq \ell}.\alpha’$, prefixed with $\alpha$ and suffixed with $\alpha’$. From lemma 3, either $\gamma_1=\alpha.(x_i)$ or $\gamma_1=\alpha.(x_j)$ (1≤i≤p). As in case 1, $\alpha.S^{\leq \ell}.\alpha’$ is partitioned according to part (2.c) of lemma 4. The i-th sub-database (1≤i≤p) of $\alpha.S^{\leq \ell}.\alpha’$, denoted $\alpha.S^{\leq \ell}.x_i.\alpha’$, is the set of subsequences of $\alpha.S^{\leq \ell}.\alpha’$ with prefix $\alpha$ and with suffix $\gamma_1=x_i.\alpha’$. As in case 1, each sub-database is in turn recursively pruned and partitioned according to $L_1=\alpha_1-\alpha_2 \ldots -\alpha_{n-1}$-growth linear ordering.

**Case 3:** $o_0 \in \{\ast\}$. A pattern-growth direction is determined during the mining process. Then, $\alpha.S^{\leq \ell}.\alpha’$ 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 studied 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


Copyrights
Copyright for this article is retained by the author(s), with first publication rights granted to the journal.
This is an open-access article distributed under the terms and conditions of the Creative Commons Attribution license (http://creativecommons.org/licenses/by/4.0/).