JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 34, 1329-1349 (2018)
DOI: 10.6688/JISE.201809_34(5).0013

Mining and Maintenance of Sequential Patterns using a Backward Generation Framework

MING-YEN LIN[1], SUE-CHEN HSUEH[2,+] AND CHIH-CHEN CHAN[1]
[1]*Department of Information Engineering and Computer Science*
Feng Chia University
Taichung, 407 Taiwan
[2]*Department of Information Management*
Chaoyang University of Technology
Taichung, 417 Taiwan
E-mail: linmy@mail.fcu.edu.tw; schsueh@cyut.edu.tw; zesiva01@hotmail.com

Common sequential pattern mining algorithms handle static databases. Once the database updates, previous mining results would be incorrect, and we need to restart the entire mining process from scratch. Previous approaches mine patterns in a forward manner in both static and incremental databases. Considering the incremental characteristics of sequence-merging, we propose a novel methodology, called *backward mining*, to update the patterns in an incremental sequence database. Stable sequences, whose support counts remain unchanged in the updated database, are identified and eliminated from the support counting process using the backward mining methodology. We develop both the *BSpan* algorithm within the pattern-growth framework and the *BSPinc* algorithm within the Apriori-based framework for incremental discovery of sequential patterns. *BSpan* prunes all the stable sequences and their super sequences so that database projections are minimized. *BSPinc* generates candidate sequences using backward extensions and mines patterns recursively within the ever-shrinking bit-sequence space. The experimental results using both synthetic and real-world datasets show that *BSpan* and *BSPinc* work an average of 4 times faster than the well-known *IncSpan* algorithm. In comparison to re-mining, the average improvement is 6 times faster.

Keywords: incremental discovery, sequential pattern, backward mining, stable sequence, sequence merging, incremental database

1. INTRODUCTION

Mining frequent sequences is a challenging research topic due to its high complexity. Extensive research has been conducted to improve the mining efficiency of sequential patterns [1, 2, 18, 19]. The issue of incremental maintenance of sequential patterns, or called incremental sequence mining, cannot be overlooked because the database is constantly updated. In this paper, we focus on efficient algorithms for incremental discovery of sequential patterns.

A sequential pattern is a frequent sub-sequence discovered from a set of transaction-sequences, where each transaction is an *itemset* (*i.e.*, a set of items). For example, each record in a transactional database is an itemset purchased by a customer at a transaction time. Data from the same customer are sorted in ascending time order into a data sequence before mining. A data sequence *supports* a sequence if each transaction of the sequence is contained by a distinct transaction in the data sequence, sequentially with

Received January 20, 2017; revised April 21 & June 9, 2017; accepted June 11, 2017.
Communicated by Chia-Feng Juang.
[+] Corresponding author.

respect to all transactions of the sequence. The number of data sequences *supporting* the sequence is referred to as the *support count*. A *sequential pattern* (also called frequent sequence) refers to a sequence whose support count exceeding a user-specified number. For example, if a sufficient number of customers in the transactional database have the purchasing sequence of home audio, projector, and then home theater, such a sequence is a sequential pattern. Sequential pattern mining is complicated, considering the abundant combinations of potential sequences, not to mention the re-mining required when databases are updated or changed.

Common algorithms for sequential pattern mining handle static databases, which mean the data in the database will not change. Once the data change, the previous mining result will be incorrect, and we need to restart the entire mining process for the new updated sequence database. However, in practice, sequence databases are not static, and they are usually updated by appending new transactions of existing customers or data sequences of new customers. Moreover, the database often changes in small increments. For example, a retail sales database is updated each week. The sales data for the new week often represent only a small percentage of the previous year's sales data. We observe that not only the database but also the sequential patterns change in small increments. Mining the entire updated database is very time-consuming because many sequential patterns' support counts do not change after the update. If we can find the sequential pattern that remains unchanged after the update, we may free from examining the support counts of such patterns. Unfortunately, finding this type of patterns is very difficult because if one transaction is appended to an existing data sequence, many possible new patterns are generated. Although re-mining the unchanged pattern is unnecessary, the changed pattern must be re-mined. Detecting unchanged patterns as early in the process as possible could benefit incremental mining very much. If we can use previously unchanged sequential patterns to speed up incremental sequence mining, this will be a useful and efficient breakthrough.

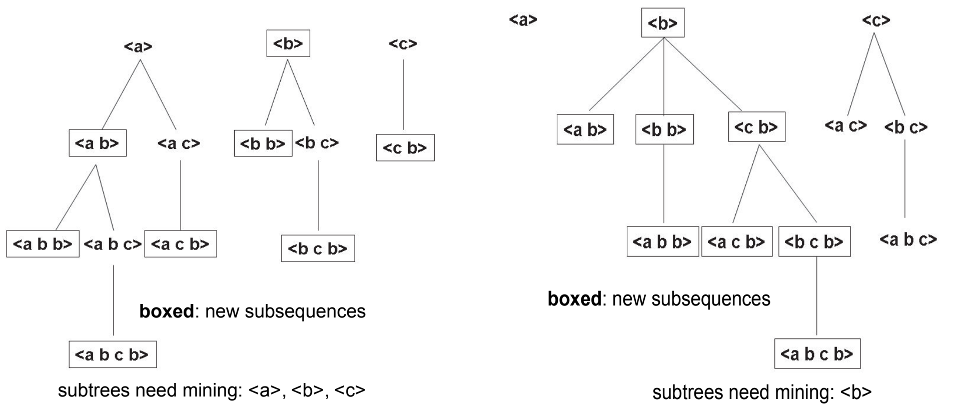
Several algorithms [3-8, 12, 17] were introduced to deal with the problem of incremental mining of sequential patterns, based on common sequence mining algorithms. However, these algorithms still have some limitations on finding the unchanged patterns for incremental mining. Previous sequential pattern mining approaches, within either the Apriori-based [2, 19, 22] or the projection-based [16, 18, 20] framework, mine patterns in a forward manner, called *forward mining* here. Let k-pattern be a sequential pattern with k items. A discovered k-pattern is used as a *prefix*, and one potential item is added *after* the k-pattern to form the candidate (k+1)-pattern in *forward mining*. For example, after mining pattern <(a)(b)> in forward mining, the pattern is employed as a *prefix* and one item *after* the pattern is added (or projected), such as <(a)(b)**(c)**>, to be tested.

After an in-depth study of the incremental characteristics of sequences, we develop a novel methodology, called *backward mining*, for efficient incremental sequence discovery. In contrast to forward mining, the candidate pattern <**(c)**(b)(a)> is mined after <(b)(a)> is discovered in backward mining. A discovered k-pattern is used as a *postfix*, and one potential item is added *before* the k-pattern to form the candidate (k+1)-pattern in the proposed backward mining methodology.

We found that backward mining methodology is more efficient than forward mining for incremental sequence discovery. A particular issue in incremental sequence mining, named *sequence merging*, is that all transactions of a customer, either appeared in previous mining or just emerged in current mining, are to be sorted into one data sequence. Consequently, newly appended items will change the previous sequence into a new sequence due to sequence merging so as to complicate the support counting in incremental sequence mining. Many new sub-sequences are generated but only the supports of those sub-sequences not existing in the previous data sequence should be increased. The merge step may introduce redundant counting operations without systematic checking. For example, a data sequence <(a)(b)(c)> appended item *b* to become <(a)(b)(c)**(b)**> so that the boxed nodes in Fig. 1 will be affected. Using backward mining, we may easily skip the checking required in forward mining, *i.e.* those nodes prefixed by items *a* and *c* in Fig. 1 (a). In addition, the anti-monotone property, referring to *all the sub-patterns of a frequent pattern must be frequent*, still holds for backward mining. Therefore, if a *k*-sequence is infrequent, none of its super sequence can be frequent. Hence, if <(c)(b)> is infrequent, <**(b)**(c)(b)>, <**(a)**(c)(b)>, <**(b)(a)**(c)(b)>, and so on can be eliminated in backward mining since they cannot be frequent. Fig. 1 shows two different methodology of sequential pattern mining.

To the best of our knowledge, no algorithms with backward mining flow have been proposed for the mining and the maintenance of sequential patterns. In this paper, we present the backward mining methodology and describe a unique property, called *stable sequence* properties, for incremental discovery of sequential patterns. We develop two incremental mining algorithms within different frameworks to show the applicability of the backward mining methodology. First, the *BSpan* (**B**ackward **S**equential **PA**tter**N** mining and updating) algorithm utilizes the *PrefixSpan* algorithm [18] to incrementally mine and update sequential patterns within the pattern-growth framework. Second, the *BSPinc* (**B**ackward **SP**AM for **inc**remental mining) algorithm utilizes the *SPAM* algorithm [2] as the basis to incrementally mine the sequential patterns within the Apriori-based framework. The extensive experiments show that the backward methodology is several times faster than the well-known incremental mining algorithms.

The rest of the paper is organized as follows. Section 2 addresses the problem statements. Section 3 briefly reviews the related work. Section 4 introduces the proposed methodology of backward mining and two algorithms *BSPinc* and *BSpan*. The Experimental results are reported and discussed in Section 5. Section 6 concludes the study.



(a) Forward mining: scattered subsequences. (b) Backward mining: grouped subsequences.
Fig. 1. Various flows of sequential pattern mining: appending **b** to <abc>.

2. PROBLEM STATEMENTS

The problem of incremental discovery of sequential patterns is formally defined as follows. Let $\Omega = \{\sigma_1, \sigma_2, \ldots, \sigma_r\}$ be a set of literals, called *items*. An *itemset* $e = (\alpha_1, \alpha_2, \ldots, \alpha_q)$ is a nonempty set of q items such that $e \subseteq \Omega$. A *sequence* s, denoted by $<e_w e_{w-1} \ldots e_2 e_1>$, is an ordered list of w *elements* where each *element* e_i is an itemset. Without loss of generality, we assume the items in an element are in **reverse** lexicographic order. The *size* of a sequence s, written as $|s|$, is the total number of items in all the elements in s. Sequence s is a *k-sequence* if $|s| = k$. A sequence $s = <e_w e_{w-1} \ldots e_2 e_1>$ is a *subsequence* of another sequence $s' = <e_m' e_{m-1}' \ldots e_2' e_1'>$ if there exist $1 \leq i_1 < i_2 < \ldots < i_w \leq m$ such that $e_1 \subseteq e_{i_1}'$, $e_2 \subseteq e_{i_2}'$, …, and $e_w \subseteq e_{i_n}'$. Sequence s' is a *super-sequence* of s if s is a subsequence of s'. Note that the parentheses for an itemset are omitted whenever there is no ambiguity. For example, <(c)(d,b)(a)> is a *4*-sequence, which can be represented as <c(d,b)a>, and it contains a *3*-sequence <cba>.

The sequence database *DB* contains |*DB*| data sequences, where each *data sequence ds* is a sequence with a unique identifier *sid*. The *support count* of sequence s, denoted by *s.count*, is the number of data sequences containing s. The *support* of sequence s, denoted by *s.sup*, is *s.count* divided by |*DB*|. The *minsup* is the user specified minimum support threshold. A sequence s is a *frequent sequence*, or called *sequential pattern*, if $s.sup \geq minsup$. A sequential pattern s having $|s| = k$ is referred to as a *k-pattern*. L_k is the set of all the *k-patterns*.

Given the *minsup* and the sequence database *DB*, the problem of sequential pattern mining is to discover *the set of all sequential patterns*, denoted by P^{DB}. In practice, the sequence database will be updated with new transactions after the mining process. Possible updating includes appending transactions, insertions of data sequences, *etc.* With respect to the same *minsup*, the incremental mining problem aims to find out the new set of all sequential patterns after database updating without re-mining the whole database. We describe the issue of incremental discovery by considering the problem of transaction appending first. Transaction modification can be accomplished by transaction deletion and appending. Table 1 lists the notations used in this paper.

Table 1. Notations used in this paper.

$\sigma_1, \sigma_2, \ldots, \sigma_r$	Items
$(\alpha_1, \alpha_2, \ldots, \alpha_q)$	A q-itemset, each α_i is an item.
$s = <e_w e_{w-1} \ldots e_2 e_1>$	A sequence with w element, each e_i is an item.
s.count	The *support count* of sequence *s*.
s.sup	The *support* of sequence *s*.
ds^{DB}, ds^{db}, ds^{UD}	A data sequence in DB, db, and UD respectively.
minsup	The user specified minimum support.
P^{DB}	The set of all sequential patterns in DB.
$s' = <(\boldsymbol{x}) e_w e_{w-1} \ldots e_2 e_1>$	A *sequence-extension* with item *x* to sequence *s*
$s' = <\{\boldsymbol{x}\} \cup \boldsymbol{e_w} e_{w-1} \ldots e_2 e_1>$	An *itemset-extension* with item *x* to sequence *s*
s-pj	*projection* of sequence *s*; the set of all data sequences containing *s* in *UD*.
s-end	*ending* of sequence $s = <e_w e_{w-1} \ldots e_2 e_1>$, *i.e.* e_1.
ds^{UD}*-inc*	*increment* of a data sequence ds^{UD}.
s-end-pj	*end-projection* of sequence *s*; the set of data sequences in *s-pj* whose *increment* contains *s-end*.
s-DB-pj	*DB-projection* of sequence *s*; *s-pj* \ *s-end-pj*

The **original database** *DB* is appended with a few data sequences after some time. The **increment database** *db* is referred to as the set of these newly appended data sequences. The *sids* of the data sequences in *db* may already exist in *DB*. The whole database combining all the data sequences from the original database *DB* and the increment database *db* is referred to as the **updated database** *UD*. A data sequence ds^{DB} in *DB* is updated with the data sequence ds^{db} and its corresponding data sequence in *UD* is ds^{UD}. An example *UD*, *DB*, and *db*, as shown in Table 2, will be used throughout the context. The data sequences ds_1, ds_2, ds_3, ds_4 and ds_6 are the data sequences that append new transactions. The data sequences ds_7, ds_8, and ds_9 are new data sequences.

Table 2. An *updated database UD*, its *increment database db* (sequences in bold face) and the original database *DB* (sequences without bold face).

sid	data sequence	*ending*	*increment*	P^{DB} (*ms*=1/3)	P^{UD} (*ms*=1/3)
ds_1	<fad **(f,e)**>	{f, e}	{f, e}	<a>:5, <b>:3	<a>:5, <b>:7
ds_2	<b(e,c)(b,a) **eb**>	{b}	{e, b}	<c>:3, <d>:3	<c>:3, <d>:6
ds_3	<f(b,a)(d,b) **f**>	{f}	{f}	<e>:2, <f>:4	<e>:5, <f>:5
ds_4	<facd **b**>	{b}	{b}	<ca>:2, <(ba)>:2	<fa>:3, <ab>:3
ds_5	<ca>	{a}	∅	<fa>:3, <bb>:2	<bb>:4, <eb>:3
ds_6	<e(f,e,b) **(d,b)**>	{d, b}	{d, b}	<eb>:2,<ad>:3 <fd>:3, <fad>:3	<fb>: 3, <(d,b)>:4 <ad>: 3, <fad>: 3
***ds*$_7$**	**<(d,b)>**	{d, b}	{d, b}		<fd>: 4,<be>:3
***ds*$_8$**	**<(d,b)(e)>**	{e}	{e, d, b}		
***ds*$_9$**	**<be(f,b)>**	{f, b}	{f, e, b}		

3. RELATED WORK

The basis of incremental mining is traditional sequential pattern mining. For example, IncSpan algorithm [5] is based on PrefixSpan [18]; PBIncSpan algorithm [4] is also based on PrefixSpan [18]; IncSP [12] extends GSP algorithm [19]; CISpan [21] uses CloSpan [20] as the base mining algorithm. Nevertheless, these approaches use the forward mining methodology. Some incremental sequence mining algorithms are reviewed briefly here.

IncSpan [5] keeps the semi-frequent patterns to speed up the discovery of the newly appeared patterns, which were not frequent in previous mining. The algorithm sets a semi-frequent ratio and uses PrefixSpan to mine not only the frequent sequential patterns but also the semi-frequent sequential patterns. It also can directly add the support count that was added by a new sequence and newly appended item. However, to have the semi-frequent patterns, a lowered support is used in the mining so that much time is spent on the mining.

PBIncSpan algorithm [4] constructs a prefix tree for a sequence database using a similar method as the mining in PrefixSpan. A width-pruning and a depth-pruning strategies are used to maintain the sequence tree after scanning the incremental part of the updated database. By checking sequence ids in the projected database of a node *p*, the width-pruning strategy shrinks the search space of a node *p* and its subtree. By checking the intersection of the projected database of a node *p* and the incremental part of the updated database, and then the intersection of the incremental element set (IES) of node *p* and its sibling nodes, the depth-pruning shrinks the search space using the Apriori property if the intersection is an empty set. However, the width-pruning needs to maintain the set of sequence ids for projected databases. The existence checking of ids is time-consuming when the updated database is huge. Especially, when the prefix tree has a large number of nodes, the depth-pruning might be inefficient with respect to the Apriori property.

IncSP algorithm [12] is an incremental mining algorithm based on the GSP algorithm [19], which is a multiple-pass candidate-generation-and-test algorithm. IncSP divides the incremental mining into two parts: one is updating the sequential patterns in the DB, and the other is the finding of the sequential patterns in the incremental database (db). Then, the algorithm merges the two sets of patterns to generate new candidates. The process is also time-consuming when the number of candidates is large.

CISpan algorithm [21] treats updating, appending, and modifications as steps of deleting old sequences and adding new ones. For example, the updating of sequence <abcd> into <abbf> is handled by deleting one sequence <abcd> and inserting one new sequence <abbf>. Nevertheless, appending itemsets to existing sequences may generate too many sequences, which need to be processed. In the worst case, the entire mining process still has to be performed.

4. BACKWARD MINING AND MAINTENANCE OF SEQUENTIAL PATTERNS

The performance and result of mining sequential patterns in static databases using either forward or backward approaches are the same. Nevertheless, mining sequential patterns in incrementally updated databases using the backward approaches significantly outperforms that using forward approaches. In this section, we present the backward mining approaches for incremental discovery of sequential patterns. Section 4.1 describes the terminology used in the backward mining. Section 4.2 addresses the stable sequence property for incremental mining using the backward mining methodology. Section 4.3 presents the proposed backward mining approach within the pattern-growth based framework. Section 4.4 describes the proposed backward mining approach within the Apriori-based framework.

4.1 Terminology used in Backward Mining

The definitions of basic terms in backward mining are the same as those stated in Section 2. The terms used particularly in the backward mining are defined as follows.

The items in an element and the elements in a sequence are defined in a reverse order, as described in Section 2. The formation of a sequence is extended backward in backward mining within both the Apriori-based and the pattern-growth based frameworks. Thus, a sequence-extension with item x to a sequence s = $<e_w e_{w-1} \ldots e_2 e_1>$ forms s' = $<(x)e_w e_{w-1} \ldots e_2 e_1>$. An itemset-extension with item x to a sequence s = $<e_w e_{w-1} \ldots e_2 e_1>$ forms s' = $<(x)e_w e_{w-1} \ldots e_2 e_1>$. Both sequence-extension and itemset-extension are extensions of s. For example, we extend sequence <ba> with an itemset <c> to form the sequence <cba>; we extend sequence <ba> with item c to the itemset (b) in <ba> to form the sequence <(c,b)a>.

The projection of a sequence *s*, denoted by *s-pj*, is the set of all data sequences containing *s* in UD. The size of the projection, written as |*s-pj*|, is the total number of data sequences in *s-pj*. The ending of a sequence *s* = $<e_w e_{w-1} \ldots e_2 e_1>$, denoted by *s*-end, refers to the element e_1. The increment of a data sequence ds^{UD}, denoted by ds^{UD}-inc, is the set of items in the corresponding ds^{db}. The increment-union of a set of data sequences is the union of the increments of all the data sequences in the set. The end-projection of a sequence *s*, denoted by *s-end-pj*, is the set of data sequences in *s-pj* whose increment contains the ending of *s*. The DB-projection of a sequence *s*, denoted by *s-DB-pj*, is the set-theoretic difference of *s-pj* and *s-end-pj*, *i.e. s-pj\s-end-pj*. In short, projections, end-projections, and DB-projections contain set of data sequences; endings, increments, and increment-unions contain set of items.

Table 3. Some end-projections and DB-projections.

sequence	end-projection	DB-projection	sequence	end-projection	DB-projection
<a>	∅	$\{ds_1, ds_2, ds_3, ds_4, ds_5\}$	<fb>	$\{ds_4, ds_6\}$	$\{ds_3\}$
<b>	$\{ds_2, ds_4, ds_6, ds_7, ds_8, ds_9\}$	$\{ds_3\}$	<(d,b)>	$\{ds_6, ds_7, ds_8\}$	$\{ds_3\}$
<ab>	$\{ds_2, ds_4\}$	$\{ds_3\}$	<(f,b)>	$\{ds_9\}$	∅
<bb>	$\{ds_2, ds_6, ds_9\}$	$\{ds_3\}$	<bab>	$\{ds_2\}$	∅
<cb>	$\{ds_2, ds_4\}$	∅	<eab>	$\{ds_2\}$	∅
<db>	$\{ds_4\}$	∅	<fab>	$\{ds_4\}$	$\{ds_3\}$
<eb>	$\{ds_2, ds_6, ds_9\}$	∅	<(b,a)b>	$\{ds_2\}$	$\{ds_3\}$

For example, the projection of <d>, <d>-*pj*, is $\{ds_1, ds_3, ds_4, ds_6, ds_7, ds_8\}$ and <fd>-*pj* is $\{ds_1, ds_3, ds_4, ds_6\}$ in Table 2. The ending of ds_1 = <fad (f,e)> is {f, e}; <b(e,c)(b,a)e b>-end (ds_2-end) is {b} and <f(b,a)(d,b) f>-end (ds_3-end) is {f}. The increment of ds_1 is {f, e}; ds_2-inc is {e, b} and ds_3-inc is {f}. The increment-union of UD is {b, d, e, f}. To obtain the end-projection of <d>, we check the increment of each sequence in the projection of <d> (<d>-pj) to find out which contains the ending of <d>. The increment of ds_1, ds_3, ds_4, ds_6, ds_7, and ds_8 is checked to see which contains {d}. Thus, we have <d>-end-*pj* = $\{ds_6, ds_7, ds_8\}$. Similarly, we find <fd>-end-*pj* to be $\{ds_6\}$. The DB-projection of <d>, <d>-DB-pj, is $\{ds_1, ds_3, ds_4\}$ and <fd>-DB-*pj* is $\{ds_1, ds_3, ds_4\}$. Table 3 lists some end-projections and DB-projections with respect to the UD in Table 2.

4.2 Stable Sequences in Backward Mining

A unique characteristic in the backward mining approaches is the identification of the stable sequence. A sequence *s* is defined as ***stable*** if the *end-projection* of *s* is an empty set. In other words, *s* is a stable sequence if *s-end* is not contained in the *increment* of any data sequence within *s-pj*. One may find the projection of *s* first. For every data sequence in the projection, if the *increment* of the data sequence does not contain *s-end*, one may determine that *s* is *stable*. Take Tables 2 and 3 for example, <a>-*end* is {a} and <a>-*pj* is $\{ds_1, ds_2, ds_3, ds_4, ds_5\}$, whose *increment-union* is {b, e, f}. Sequence <a> is stable since {a} $\not\subset$ {b, e, f}. Considering <(b, a)>, which is an extension of <a>. <(b, a)>-*end* is {a, b}, <(b, a)>-*pj* is {ds_2, ds_3}, whose *increment-union* is {b, e, f}. {a, b} $\not\subset$ {b, e, f} so that sequence <(b, a)> is stable. Sequences <a>, <(b, a)>, <ca>, <fa>, <c>, <ad>, and so on are stable.

The stable sequence owns two important properties to be used in the backward mining for the substantial improvements on incremental sequence mining.

Property 1: The support of a stable sequence in *UD* is the same as that in *DB*.

Proof: Let ds^{UD} be a data sequence in the projection of *s* and the *increment* of ds^{UD} be *M*, which is the set of items in ds^{db}. Let the *ending* of *s* be *N*, which is the set of items in e_1 of *s*. ds^{UD} contains *s* since ds^{UD} comes from *s-pj*. If *M* does not contain *N* then *N* must be contained in ds^{DB}. Because the *increment* of every data sequence in *s-pj* does not contain the *ending* of *s*, the support counts of *s* in *UD* are "contributed" by all the corresponding data sequences in *DB*. Therefore, the support count of a stable sequence *s* in *UD* and that in *DB* are the same. ❑

Property 2: Any extensions of a stable sequence must be stable.

Proof: Let *s*′ be an extension of *s*. The *s*′*-end* is the same as *s-end* since both *itemset-extension* and *sequence-extension* extend *s* backward. The *s*′*-pj* is a subset of *s-pj* so that *s*′*-end-pj* is a subset of *s-end-pj*. The *s-end-pj* is an empty set if *s* is stable, by definition. Thus, *s*′ is stable because *s*′*-end-pj*, which is a subset of *s-end-pj*, is an empty set. ❑

Note that property 1 addresses that the support count of a stable sequence remains the same in the updated database. Therefore, we may skip the support counting process of stable sequences in backward mining approaches. Furthermore, property 2 identifies the group of stable sequences, generated by the proposed backward extensions for super-sequence formations. As a result, the stable sequences and all their extensions can be eliminated from the support counting process within both the Apriori based and the pattern-growth based framework.

The stable sequence property is utilized in the proposed approaches to speed up the mining extensively. The experimental results in Section 5 indicate that a large number of projected databases are eliminated in *BSpan*. Also, many candidate patterns in *BSPinc* were pruned. More than 75% patterns were pruned, when both algorithms incrementally mined the synthetic dataset C10-T2.5-S4-I1.25 with minimum support 0.001 and modification ratio 0.1.

4.3 BSpan: Backward PrefixSpan for Incremental Mining

In this section, we describe the proposed incremental sequential pattern mining algorithm within the *PrefixSpan* framework using the backward mining methodology. The algorithm is called *BSpan* (**B**ackward **S**equential **PA**tter**N** mining and updating). We describe the terms used particularly in the *BSpan* algorithm as follows.

A sequence $s' = \langle e_v' e_{v-1}' \ldots e_2' e_1' \rangle$ is a ***postfix*** of sequence $s = \langle e_w \ldots e_2 e_1 \rangle$ ($v \le w$) if and only if the following conditions all hold: (1) $e_j' = e_j$ for $1 \le j \le v-1$; (2) $e_v' \subseteq e_v$; (3) $\forall$ item $x \in (e_v \setminus e_v')$ and $\forall y \in e_v'$, x is lexicographically after y. Sequence $s'' = \langle e_w e_{w-1} \ldots$

$e_{v+1}e_v''$> is called the ***prefix*** of *s* with respect to postfix *s'* where $e_v'' = e_v \setminus e_v'$. For example, <fa> is the prefix of <**fa** d(f,e)> with respect to postfix <d(f,e)>; <e(f,e_)> is the prefix of <e(f,e,**b)(d,b)**> with respect to postfix <b(d,b)>. The (f,e_) means that it is an item-set-extension of the postfix sequence. The ***projected database*** of a sequence *s* is the set of **prefixes** of all the data sequences in *UD* with respect to postfix *s*, *denoted by* $UD|_s$. The ***increment-projected database*** of a sequence *s*, denoted by *s-end-pj*$|_s$, is the set of prefixes of all the data sequences in the *end-projection* of *s* (*s-end-pj*). The ***original-projected database*** of a sequence *s*, denoted by *s-DB-pj*$|_s$, is the set-theoretic difference of the *projected database* and the *increment-projected database*. In short, projected databases, increment-projected databases, and original-projected databases contain **set of prefixes** from the updated databases.

For example, given *UD* in Table 2, the *projected database* of <d>, $UD|_{<d>}$, is the set of prefixes from *<d>-pj* = {ds_1, ds_3, ds_4, ds_6, ds_7, ds_8}, *i.e.* $UD|_{<d>}$ = {<fa>, <f(b,a)>, <fac>, e(f,e,b)} since the prefixes of both ds_7 and ds_8 are null. The *increment-projected database* of <d>, *<d>-end-pj*$|_{<d>}$, is {<e(f,e,b)>} and the *original-projected database* of <d>, *<d>-DB-pj*$|_{<d>}$, is {*<fa>*, <f(b,a)>, <fac>}. $UD|_{<fd>}$ is {<e>}, which is the prefix of ds_6. *<fd>-end-pj*$|_{<fd>}$ = {<e>}, and *<fd>-DB-pj*$|_{<fd>}$ is ∅. Table 4 shows some projected databases of the example updated database in Table 2.

We use *PrefixSpan* [18] as the fundamental mining framework, enhance it with backward mining, and propose the *BSpan* algorithm for incremental sequence mining. We modify the pseudo projection version of the *PrefixSpan* algorithm [18] by projecting databases with respect to postfixes instead of prefixes. The *BSpan* algorithm, like *PrefixSpan*, counts (*k*+1)-*sequences* from *k-patterns*' *projection*. A projected database is divided into the *increment-projected database* and the *original-projected database*. If the *increment-projected database* of a sequence is an empty set, the sequence is stable and can be eliminated from projection and counting. A sequence having non-empty *increment-projected database* is not stable and a *projected database* would be generated for the subsequent mining process.

Table 4. Some *increment-projected databases* and *original-projected databases*.

sequence	*increment-projected database* (*s-end-pj*$\vert_s$)	*original-projected database* (*s-DB-pj*$\vert_s$)
<a>	∅	{<f>,*<b(e,c(b_)>*,*<f(b_)>*, *<f>*, *<c>*}
<b>	{<b(e,c)(b,a)e>, *<facd>*,*<e(f,e,b)(d_)>*, *<(d_)>*, *<(d_)>*, *<be(f_)>*}	{<f(b,a)(d_)>}
<ab>	{<b(e,c)(b_)>, *<f>*}	{<f(b_)>}
<bb>	{<b(e,c)>,*<e(f,e_)>*}	{<f>}
<cb>	{<b(e_)>, *<fa>*}	∅
<db>	{*<fac>*}	∅
<eb>	{<b(e,c)(b,a)>, *<e(f,_)>*, *<b>*}	∅
<fb>	{*<e>*}	{∅}
<(d,b)>	{*<e(f,e,b) >*}	{<f(b,a)>}
<(f,b)>	{*<be>*}	∅
<bab>	∅	∅
<eab>	{<b>}	∅
<fab>	∅	∅
<(b,a)b>	{<b(e,c)>}	{<f>}

Fig. 2 presents the *BSpan* algorithm and Fig. 3 shows the subroutine in the *BSpan* algorithm. The *BSpan* algorithm first scans *UD* to get all the frequent items as 1-patterns and obtains the *increment-union* of *UD* at the same time. If an item *x* of a 1-pattern is not in the *increment-union*, clearly *x* appears in *DB* and the 1-pattern is a stable sequence. The *BSpan* algorithm can skip the support counting step of stable sequences because the support count of a stable sequence in *UD* is the same as that in *DB*. Such a skip-counting technique is referred to as *stable sequence pruning*. Moreover, a unique property is introduced by backward mining in Section 4.2: any extensions of a stable sequence must be stable. Thus, once a stable sequence *s* is discovered, the support counting step of all extensions of *s* can be skipped. The stable sequence pruning can significantly speed up the incremental mining process. To determine whether a stable sequential pattern *s* is still frequent after updating, we can simply check its support count, which is available without any computations, against the new minimum support count in *UD*, *i.e. minsup**|*UD*|. When a 1-pattern exists in the increment-union, the *BSpan* algorithm generates its projected database as the *increment-projected database* and the *original-projected database* for the subsequent mining.

```
Algorithm: BSpan
Input: DB (a sequence database before update), UD (a sequence database after update),
minsup (minimum support), $P^{DB}$ (previous sequential patterns in DB)
Output: $P^{UD}$ (sequential patterns in UD)
1. scan UD to get $L_1$, the set of 1-patterns, and the increment-union of UD. $P^{UD} = L_1$.
2. for each 1-pattern <x> in $L_1$ and item $x \notin$ increment-union of UD do /* <x> is stable */
3.     for each extension of <x>, s', do
4.         get s'.count from $P^{DB}$, if $s'.count \geq minsup*|UD|$ then $P^{UD} = P^{UD} \cup \{s'\}$.
5.     endfor
6. endfor
7. for each 1-pattern <x> in $L_1$ and item $x \in$ increment-union of UD do /* <x> is not stable */
8.   generate <x>-DB-pj$|_{<x>}$ and <x>-end-pj$|_{<x>}$
9.   call Postfix_proj (<x>, <x>-end-pj$|_{<x>}$, <x>-DB-pj$|_{<x>}$).
10.endfor
```
Fig. 2. Algorithm *BSpan*.

The *BSpan* algorithm determines whether a *k*-sequence *s* ($k > 1$) is stable by checking its *increment-projected database*, *s-end-pj*$|_s$. If *s-end-pj*$|_s$ is an empty set, *s* is stable so that *s* and all its extensions are eliminated from counting. Furthermore, the search space is shrunk iteratively during mining because if *s'* is an extension of *s*, *UD*$|_{s'}$ is a subset of *UD*$|_s$ and *s-end-pj*$|_{s'}$ is a subset of *s-end-pj*$|_s$. We may skip the checking of the extensions of *s* once *s-end-pj*$|_s$ is an empty set.

Subroutine *Postfix_proj* is used for the support counting in increment-projected databases. It first verifies whether a sequence is stable. If a sequence is found to be stable, its support count can be obtained from P^{DB} (sequential pattern in *DB*) directly. Moreover, all extensions of the stable sequences are stable so that we may immediately determine whether its extensions are frequent from P^{DB}. Consequently, no projected databases are generated.

```
Subroutine Postfix_proj ($p$, $p\text{-}end\text{-}pj|_p$, $p\text{-}DB\text{-}pj|_p$)
Input: $p = <e_k \dots e_2e_1>$, $p\text{-}end\text{-}pj|_p$ is the increment-projected database of $p$, $p\text{-}DB\text{-}pj|_p$ is
the original-projected database of $p$
1. scan $p\text{-}end\text{-}pj|_p$ and $p\text{-}DB\text{-}pj|_p$ and find each item $b$, such that $b$ can be used in a se-
     quence-extension or an itemset-extension. /* $<b>$.count $\geq$ $minsup$*$|UD|$ */
2. for each frequent item $b$ do
3.   extend $p$ with $b$ to form pattern $p'$. $P^{UD} = P^{UD} \cup p'$.
4.   if the count of $b$ in $p\text{-}end\text{-}pj|_p = 0$ then /* $p'$ is a stable sequence */
5.       for each sequence $s'$, where $s'$ is an extension of $p'$, $s' \in P^{DB}$ do
6.         get $s'.count$ from $P^{DB}$, if $s'.count \geq minsup$*$|UD|$ then $P^{UD} = P^{UD} \cup \{s'\}$.
7.     endfor
8.   else /* $p'$ is not a stable sequence */
9.       generate $p'\text{-}DB\text{-}pj|_{<x>}$ and $p'\text{-}end\text{-}pj|_{<x>}$
10.        call Postfix_proj ($p'$, $p'\text{-}end\text{-}pj|_{p'}$, $p'\text{-}DB\text{-}pj|_{p'}$).
11.  endif
12. endfor
```
Fig. 3. Subroutine *Postfix_proj*.

We check the support count of a sequence s' in increment-projected database: if this value is 0 then s' is stable. The support count in the increment-projected database of sequence s' can be obtained from counting in the increment-projected database of s, where s is the postfix of s'. The support count of an unstable sequence is obtained from the *original-projected database of* s'. *Postfix_proj* performs support counting and stable sequence detection in the same time.

Note that the minimal support count changes because $|UD|$ changes. Although the support count of a stable sequence is the same, the pattern has to be validated against the new minimum count. We generate *original-projected databases* for an unstable sequence to get its support count. The final support count of an unstable sequence s is the sum of $|s\text{-}end\text{-}pj|_s|$ and $|s\text{-}DB\text{-}pj|_s|$.

When the database is incrementally updated, *PrefixSpan* has to re-mine the whole database for the up-to-date patterns, while *BSpan* utilizes the stable sequence properties and discovers the sequential patterns efficiently.

An illustrating example of running *BSpan* on the updated database *UD* (*minsup* = 1/3) in Table 2 is given below.

Running BSpan: scan *UD* once, we have the set of 1-patterns {<a>:5, <b>:7, <c>:3, <d>:6, <e>:5, <f>:5} and the increment-union {b, d, e, f}. Thus, sequences <a> and <c> are stable. We can skip the support counting and database projection for sequences <a> and <c>. In addition, we can determine whether any extensions of <a> and <c>, if they exist in P^{DB}, are frequent by validating their support counts in P^{DB}. Sequence <ca> (also <(b,a)>) is no longer frequent in *UD* since its support count is 2 but <fa> remains frequent in *UD*. We also know that <b>, <d>, <e>, and <f> are not stable. Therefore, the following steps are performed on 1-patterns <b>, <d>, <e>, and <f> subsequently. Algorithm *BSpan* calls *Postfix_proj* on <b>, *i.e. Postfix_proj*(*<b>*, *<b>-end-pj* $|_{<b>}$, *<b>-DB-pj*$|_{<b>}$), on <d>, on <e>, and on <f>. The operations of *Postfix_proj*(*<d>*, *<d>-end-pj*$|_{<d>}$, *<d>-DB-pj*$|_{<d>}$) are described below.

Running Postfix_proj: we have <*d*>*-end-pj*$|_{<d>}$ = {<e(f,e,b)>} and <*d*>*-DB-pj*$|_{<d>}$ = {<fa>,<f(b,a)>,<fac>}. After scanning <*d*>*-end-pj*$|_{<d>}$ and <*d*>*-DB-pj*$|_{<d>}$, we have {a:0+3, b:1+1, c:0+1, e:1+0, f:1+3}. The a:0+3 means the count of item *a* in <*d*>*-end-pj* is 0 and that in <*d*>*-DB-pj* is 3. Sequential patterns <ad>:3 and <fd>:4 will be added to P^{UD}. Since the count of item *a* in <*d*>*-end-pj* is 0, <ad> is a stable sequence. Thus, <fad> is also a stable sequence and we may add <fad> to p^{UD} by obtaining its count of 3 directly from p^{DB} without generating any projections for counting. We need to generate <*fd*>*-end-pj*$|_{<fd>}$ and <*fd*>*-DB-pj*$|_{<fd>}$ for the recursive call of *Pofix_proj* on <fd>. It turns out that no sequential patterns are generated from the extensions. Consequently, the sequential patterns generated by *Postfix_proj*(<*d*>, <*d*>*-end-pj*$|_{<d>}$, <*d*>*-DB-pj*$|_{<d>}$) are {<ad>:3, <fad>:3, <fd>:4}.

Take *Postfix_proj*(<*b*>, <*b*>*-end-pj*$|_{<b>}$, <*b*>*-DB-pj*$|_{<b>}$) for another example. We have <*b*>*-end-pj*$|_{<b>}$ = {<b(e,c)(b,a)e>, <facd>, <e(f,e,b)(d_)>, <d_>,<d_>, <be(f_)>} and <*b*>*-DB-pj*$|_{<b>}$ = {<f(b,a)(d_)>}. After scanning the two projections, we have {a:2+1, b:3+1, c:2+0, d:1+0, e:3+0, f:2+1, d_:3+1, f_:1+0}. Sequential patterns <ab>:3, <bb>:4, <eb>:3, <fb>:3, <(d,b)>:4 will be added to p^{UD} and they will perform recursive extensions by calling *Postfix_proj* each. All these calls return no patterns.

Finally, we have the sequential patterns in *UD* P^{UD} = {<a>:5, <b>:7, <c>:3, <d>:6, <e>:5, <f>:5, <fa>:3, <ab>:3, <bb>:4, <eb>:3, <fb>:3, <ad>:3, <fd>:4, <fad>:3, <(d,b)>:4, <be>:3}.

4.4 BSpan: Backward PrefixSpan for Incremental Mining

The proposed incremental mining algorithm, *BSPinc* (**B**ackward ***SP**AM* for **inc**remental mining), is within the *Aprioi* framework using the backward mining methodology. We adapt the *SPAM* algorithm [2] with the primary modification of extending candidate patterns *in a backward manner*. When the database is incrementally updated, *BSPinc* utilizes the stable sequence properties and discovers the sequential patterns efficiently.

The bitmapped representation in [2] is also used in the *BSPinc* algorithm. The set of 1-*patterns* are obtained after scanning *UD* once. Each 1-pattern is then extended to form the candidate sequences for further testing. The *BSPinc* algorithm, like the *SPAM* algorithm, generates candidate (*k*+1)-*sequences* from *k-patterns*. The candidate patterns are then counted to determine whether they are frequent. Both sequence-extensions and itemset-extensions in candidate generations are conducted in a **backward** manner. The candidate generation-and-test, improved with our backward mining and stable sequence pruning, is performed until no more candidate is generated. Fig. 4 shows an example of candidate generation using items *a* and item *b*, up to the candidate 3-sequences.

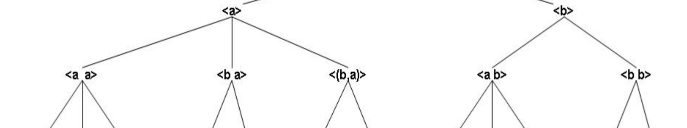


Fig. 4. Backward candidate generation: using items *a* and *b* for example.

For example in Table 2, we may scan <d>-*end-pj*, which is {ds_6, ds_7, ds_8} to get <ad>-*end-pj*, which is ∅. Thus, <ad> and all its extensions are stable so that further counting for these sequences are not required. For convenience, some end-projections and DB-projections are listed in Table 3.

Fig. 5 outlines the *BSPinc* algorithm. The subroutine *SS_pruning* used in *BSPinc* algorithm is shown in Fig. 6. *BSPinc* first scans *UD* to get all 1-patterns, their *end-projections*, and their *DB-projections.* The *increment-union* of *UD* is obtained in the same time. All the 1-patterns whose item disappears in the *increment-union* are stable sequences. Except the stable sequences, the 1-patterns are to be backward extended, by the proposed sequence-extensions and itemset-extensions, and recursively mined to determine whether the extensions are frequent in *UD*.

```
Algorithm: BSPinc
Input: DB (a sequence database before update), UD (a sequence database after update),
minsup (minimum support), $P^{DB}$ (previous sequential patterns in DB)
Output: $P^{UD}$ (sequential patterns in UD)
1. scan UD to get $L_1$, the set of 1-patterns, DB-projections of $L_1$, end-projections of $L_1$,
   and increment-union of UD. $P^{UD} = L_1$.
2. for each 1-pattern <x> in $L_1$ and item $x \notin$ increment-union of UD do /* <x> is stable */
3.    for each sequence s', where s' is an extension of <x> do
4.        get s'.count from $P^{DB}$, if $s'.count \geq minsup*|UD|$ then $P^{UD} = P^{UD} \cup \{s'\}$.
5. for each 1-pattern <x> in $L_1$ and item $x \in$ increment-union of UD do /* <x> not
   stable */
      /* $\{y| y \leq x\}$ is the set of items which are lexicographically before item x */
6.    call SS_Pruning (<x>, $L_1$, $L_1$-$\{y| y \leq x\}$).
```
Fig. 5. Algorithm *BSPinc*.

Subroutine *SS_pruning*, abbreviated from *stable sequence pruning*, is used to verify whether a candidate sequence is stable first. If a candidate is found to be stable, its support count can be obtained from P^{DB} without further support counting in *UD*. Moreover, any super sequence of the candidate is stable so that we may immediately determine whether its super sequence is frequent from P^{DB}. We check the end-projection of a candidate sequence *s*′: if *s*′-*end-pj* = ∅ then *s*′ is stable. The *s*′-*end-pj* can be obtained by scanning *s-end-pj*, where *s* is the postfix of *s*′. The support count of an unstable sequence is obtained by scanning its DB-*projection*.

Table 5. Total execution time of dataset C10-T2.5-S4-I1.25, minsup=0.02.

Modification ratio	*IncSP*	*BSpan*	*BSPinc*
10%	2.671(s)	0.031(s)	0.042(s)
20%	3.016(s)	0.034(s)	0.051(s)
40%	3.391(s)	0.041(s)	0.062(s)

```
Subroutine SS_Pruning (p, Sc, Ic)
Input: $p = <e_k \ldots e_2 e_1>$, Sc = the set of candidate items to be used in sequence-extensions,
Ic = the set of candidate items to be used in itemset-extensions
/* Ssc = the set of stable candidate items in sequence-extensions */
/* Isc = the set of stable candidate items in itemset-extensions */
1. $Sc' = Ic' = Ssc = Isc = \varnothing$.
2. for each item sq in Sc do /* sequence-extension */
3.     $s' = <\{\underline{sq}\}\ e_k \ldots e_2 e_1>$, scan p-end-pj to get s'-end-pj.
4.  if s'-end-pj = $\varnothing$ then /* s' is stable */
5.      $Ssc = Ssc \cup \{sq\}$.   /* SS_pruning is not required for item sq */
6.      get s'.count from PDB.
7.      if s'.count $\geq$ minsup*|UD| then PUD = PUD $\cup$ {s'}; $Sc' = Sc' \cup \{sq\}$.
8.      for each extension s" of s' do
9.          get s''.count from PDB, if s''.count $\geq$ minsup*|UD| then PUD = PUD $\cup$ {s''}.
10.     else /* s' is not stable */
11.         $Sc' = Sc' \cup \{sq\}$. /* sq will be used for recursive sequence-extension */
12.         scan p-DB-pj to get s'-DB-pj,
13.         if (|s'-end-pj|+|s'-DB-pj|) $\geq$ minsup*|UD| then PUD = PUD $\cup$ {s'}.
14.    endif
15. for each item $cq \in Sc'$ and $cq \notin Ssc$
16.     call SS_Pruning ($<\{cq\}\ e_k \ldots e_2\ e_1>$, Sc', $Sc' - \{y \mid y \leq cq\}$).
17. for each item iq in Ic do /* itemset-extension */
18.     $s' = <\{iq\} \cup e_k e_{k-1} \ldots e_2 e_1>$, scan p-end-pj to get s'-end-pj.
19.     if s'-end-pj = $\varnothing$ then /* s' is stable */
20.         $Isc = Isc \cup \{iq\}$; /* SS_pruning is not required for item iq */
21.         get s'.count from PDB.
        /* if s' is frequent, iq will be used for recursive itemset-extension */
22.         if s'.count $\geq$ minsup*|UD| then PUD = PUD $\cup$ {s'}; $Ic' = Ic' \cup \{iq\}$.
23.            for each extension s'' of s' do
24.               get s''.count from PDB, if s''.count $\geq$ minsup*|UD| then PUD = PUD $\cup$ {s''}.
25. else /* s' is not stable */
26.     $Ic' = Ic' \cup \{iq\}$. /* iq will be used for recursive itemset-extension */
27.     scan p-DB-pj to get s'-DB-pj
28.           if (|s'-end-pj|+|s'-DB-pj|) $\geq$ minsup*|UD| then PUD = PUD $\cup$ {s'}.
29. for each item $cq \in Ic'$ and $cq \notin Isc$
30.     call SS_Pruning ($<\{cq\} \cup e_k e_{k-1} \ldots e_2 e_1>$, Ic', $Ic' - \{y \mid y \leq cq\}$)
```

Fig. 6. Subroutine SS_pruning of the BSPinc algorithm.

5. PERFORMANCE EVALUATION EXPERIMENTAL RESULTS

5.1 Setup of the Experiments

To assess the performance of the proposed algorithms, comprehensive experiments using a 2GHz AMD Sempron PC with 1GB memory were conducted. Both synthetic datasets and a real-world dataset were used in the experiments. The synthetic datasets were generated using the IBM dataset generator [1]. Please refer to [1] for the detail of the parameters. The real-world dataset Gazelle contains click-stream data from Gazelle.com and was processed into click sequences of customers [9]. In summary, there were 29369 data sequences, the maximum sequence size was 628, the maximum session size was 267, and the maximal session number in a data sequence was 140.

The total number of data sequences, *i.e.* |UD|, was determined and these synthetic data sequences were generated. Some data sequences are then split into two parts to simulate their counterparts in the original database and the incremental database. We used the open source from *IncSpan* [5] to cut the data sequences. The *modification ratio* is the percentage of data sequences in *UD* that will be modified (to form *DB* and *db*). For the modified data sequences, 10% more transactions of the original sequences are appended.

We implemented the *SPAM* algorithm and obtained the *PrefixSpan* from the *Illimine* website (http://illimine.cs.uiuc.edu) as the basis of common sequence mining. *SPAM* took 20.39 seconds while *PrefixSpan* took 16.1 seconds on mining dataset C10-T2.5-S4-I1.25 with *minsup* 0.001. When the support was lowered to 0.0005, *SPAM* took 24 seconds while *PrefixSpan* took 42 seconds.

5.2 Performance Comparison with IncSP

First, the proposed approaches were compared with the incremental mining algorithm *IncSP* [12], a *GSP* based algorithm. *IncSP* is a multiple pass, candidate generation-and-test algorithm for mining sequential patterns. Table 5 shows that both *BSpan* and *BSPinc* outperformed *IncSP* in total execution time. The execution time gaps were so large that *IncSP* was not compared in the rest of the experiments. The number of data sequences in *UD* was 10000.

5.3 Performance Comparison with PBIncSpan

Second, the proposed approaches were compared with the *PBIncSpan* algorithm [4]. *PBIncSpan* first constructs a prefix tree, keeps the tree in memory, then maintains the sequence tree by scanning the incremental part of the updated database. To enable the width-pruning strategy, *PBIncSpan* needs to construct the sets of sequence ids for projected databases with respect to nodes. Thus, the required memory space is very huge for *PBIncSpan*. In our implementation, *PBIncSpan* spent 1.58 seconds on mining dataset C10-T2.5-S4-I1.25 and *N*=10000 with *minsup* 0.02, for modification-ratio of 10%. The time increased to 2.87 seconds when the modification-ratio was increased into 20%. *PBIncSpan* suffers from the huge memory required for storing the prefix tree and the increment element sets (IES) of nodes for intersection. The peak memory requested by *PBIncSpan* was up to 503 MB when the modification ratio was 20%. Datasets having 10000 items makes the prefix tree to have a large number of nodes so that the mining cannot finish in a reasonable time for modification-ratio of 40%. *BSpan* outperformed *PBIncSpan* for both incremental minings.

Next, the datasets were scaled up by increasing the number of data sequences to 100K. Unfortunately, *PBIncSpan* cannot successfully completed the mining with respect to 100k sequences since the prefix tree for preserving the large database required a huge memory. Maintaining the sequence ids for width-pruning and depth-pruning also used up the memory with respect to the low supports. The experimental results are consistent with the report in [4], which describes that *PBIncSpan* has similar performance as *PrefixSpan* when the dataset is small, and that the depth-pruning might be inefficient with respect to the *Apriori* property when the prefix tree has a large number of nodes. The report in [16] also depicts that the existence checking of sequence ids is time-consuming when the updated database is huge. The stability and scalability needs to be investigated, as described in [4]. Therefore, *PBIncSpan* was not compared in the rest of the experiments.

5.4 Performance Comparison with IncSpan

The *IncSpan* algorithm [5] is a well-known algorithm for incremental sequence mining, based on a concept of semi-frequent patterns. Two optimizations including reverse pattern matching and shard projection are presented in *IncSpan*. The reverse pattern matching is totally different from our proposed backward mining, which will be discussed in Section 5.5. We compared the execution times of incremental mining using *IncSpan* and our approaches. *BSpan* and *BSPinc* are four times faster than *IncSpan* for all the modification ratios on mining dataset C10-T2.5-S4-I1.25, as shown in Fig. 7. When the modification ratio increases, the number of stable sequences decreases so that the total execution time increases. Fig. 8 depicts that the improvements of incremental mining over re-mining. It shows that incremental mining algorithms *IncSpan*, *BSpan* and *BSPinc* definitely outperform their re-mining based counterparts. *BSpan* and *BSPinc* improve much more from backward mining than *IncSpan*. Fig. 9 shows that *BSpan* and *BSPinc* consistently outperform *IncSpan* with respect to different minimum supports in total execution time. The result of mining longer data sequences is consistent, as shown in Fig. 10 on mining dataset C15-T5-S8-I2.5, |UD|=10000, |db|=1000, *N*=10000, *minsup*=0.5%.

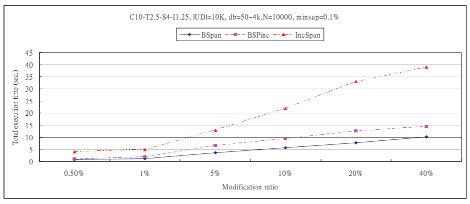


Fig. 7. Varying modification ratios.

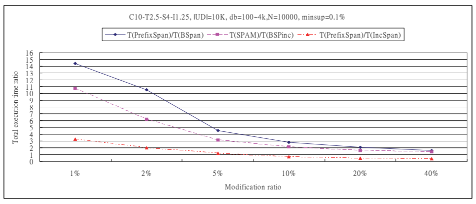


Fig. 8. Improved ratio.

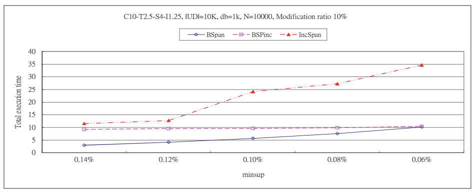


Fig. 9. Varying minimum supports.

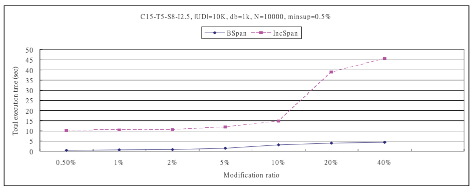


Fig. 10. Varying ratios on C15-T5-S8-I2.5.

The stable sequence property presented in this paper greatly eliminates many candidates from support counting. The total number of candidate sequences in *SPAM* for dataset C10-T2.5-S4-I1.25 and *minsup* 0.001 are 25.8 million. As shown in Fig. 11, *BSPinc* reduced more than 10 million candidate sequences. When the modification ratio is 0.5%, 25 million nodes (candidates) were pruned. Fig. 12 shows the number of pruned candidates utilizing the stable sequence property. Based on the property, their support counts in the projected database are the same as that in the original database so that many projections and support counting are eliminated. It confirms that the stable pattern pruning is very efficient in detecting patterns of unchanged support counts. When the modification ratio is 1%, almost 96% patterns were pruned. Even when the modification ratio is raised to 40%, backward mining may still prune 45% patterns. Stable sequence pruning benefits *BSpan* so that it outperforms the *IncSpan* algorithm for more than 4 times. The other algorithms in the comparisons cannot benefit from the unique property so that no numbers are shown in the figures.

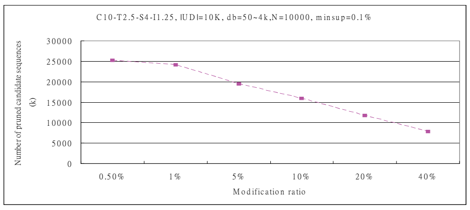


Fig. 11. Number of pruned candidates.

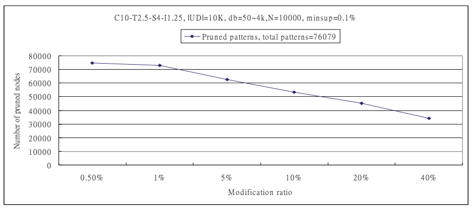


Fig. 12. Number of pruned patterns.

Both *SPAM* and *BSPinc* may consume a considerable amount of memory. The amount depends on the total number of data sequences, the total number of items, and the lengths of data sequences. Given 10000 customers, 10000 items, and sequence length of 32, it may use up to 10000*10000*32*4=400 Mbytes. The peak memory used by *SPAM* was 458 MB for dataset C10-T2.5-S4-I1.25 with *minsup* 0.001. When the modification ratio was 10%, the peak memory used by *BSPinc* was 469 MB. The memory used by *BSPinc* is too large to be shown in the comparisons. Fig. 13 shows the memory usage by *BSpan* and *IncSpan*. Only the previous patterns have to be stored and few projections are generated so that the memory usage for *BSpan* is more efficient.

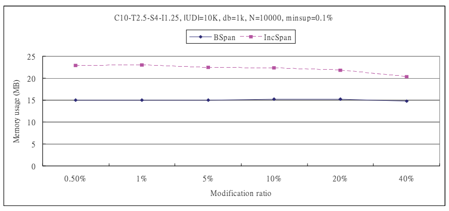


Fig. 13. Memory usage.

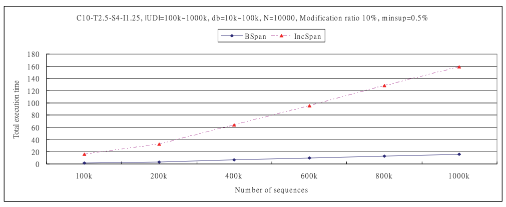


Fig. 14. Scale-up of the dataset sizes.

Fig. 14 demonstrates the results of scale-up experiments. The number of data sequences was increased from 100K to 1000K. *IncSpan* needs to scan all the data sequences and discover the semi-frequent sequential patterns, using 0.8 * *minsup* as the threshold, in *DB*. *BSpan* generates the projected databases only for the unstable sequences so that it outperforms *IncSpan*.

The Gazelle dataset contains many long data sequences and the maximum length of the sequence is 628. The bitmap representation used in *BSPinc* cannot handle such long sequences. Thus, only *BSpan* is compared with *IncSpan*. Fig. 15 shows the execution time with respect to different modification ratios for *minsup* of 0.04%. *BSpan* is 10 times faster than *IncSpan*. The long data sequences cannot be efficiently handled by *IncSpan* since a great amount of sequences are generated due to database appending. Note that when the ratio increased from 10% to 20%, unfortunately many new patterns were generated due to the appended items, so that the total mining had a large increase. The mechanism of the lowered support (described in Section 3) in *IncSpan* was effective when the ratio changed from 20% to 40%. Most of the semi-frequent patterns need no re-mining so the total execution time drops. Table 6 lists the number of patterns pruned by using the stable sequence property. *BSpan* pruned 73047/76079 = 96% of the patterns for modification ratio of 1%. Fig. 16 indicates that both *BSPinc* and *BSpan* consistently outperform *IncSpan* with respect to different minimum supports.

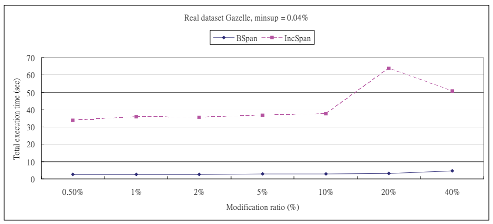


Fig. 15. Varying modification ratio.

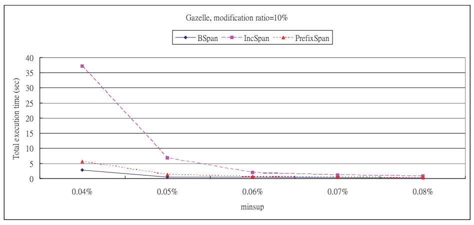


Fig. 16. Varying minimum supports.

Table 6. The number of pruned patterns in mining the Gazelle dataset.

Pattern size	Number of patterns	Number of pruned patterns, modification ratio =		
		1%	10%	40%
1	4126	3951	2984	1768
2	15778	15363	12633	8855
3	18004	17408	13609	9083
4	15445	14783	10981	6956
5	10780	10194	7062	4215
6	6493	6106	3775	2106
7	3382	3215	1658	870
8	1457	1417	563	274
9	484	480	132	55
10	113	113	18	5
11	16	16	1	0
12	1	1	0	0

5.5 Discussion: Difference Between BSpan and IncSpan

The proposed backward mining methodology might be misjudged as similar to the reverse pattern matching technique in *IncSpan* at first glance. We present the fundamental differences between the two algorithms as follows.

(1) *Pattern mining order*. *IncSpan* uses the traditional forward mining methodology to mine and detect previous sequential patterns in *DB*. A brand new backward flow is used in our methodology for the mining. The stable sequence property is uniquely held only in the backward mining methodology. In addition, the backward methodology projects fewer and smaller databases normally.

(2) *Reversal pattern matching technique*. Backward mining is totally different from the reverse pattern matching technique. The reverse matching technique is only used for checking support increase of a sequential pattern in *DB* by matching a pattern against a sequence reversely. After the testing, the database projection and support counting processes in *PrefixSpan* re-applied to the increased data sequences. The support-checking step in fact is the overhead of incremental mining. Our backward mining algorithm performs support counting and stable-sequence detection at the same time. After the item counting in the increment-projected databases, all the stable sequences are identified. Moreover, the subsequent counting and projections with respect to each stable sequence and its extensions are eliminated. As confirmed by the experiments, most of the previous patterns are stable and pruned. *IncSpan* needs to check not only all the previous patterns but also all the previous semi-frequent patterns. This is the reason why our algorithms are more efficient than *IncSpan*.

6. CONCLUSIONS

We have proposed a novel incremental mining methodology, called *backward mining*, for incremental discovery of sequential patterns. Using backward mining, the stable sequence property effectively improves the efficiency of sequence mining in incremental databases. We have designed two algorithms, *BSpan* and *BSPinc*, based on the backward mining methodology. The *BSpan* algorithm utilizes the stable sequence property to systematically eliminate a large number of unnecessary database projections. Using the same property, the *BSPinc* algorithm also enhances *SPAM* and prunes a large number of candidates. The experimental results show that both *BSpan* and *BSPinc* outperform the well-known *IncSpan* algorithm and *IncSP* algorithm. Moreover, both algorithms are four times faster than the *IncSpan* algorithm in execution time. We believe that the methodology can be used to enhance algorithms for applications requiring sequence merging such as sequential pattern mining over data streams.

ACKNOWLEDGEMENTS

The authors are grateful for the helpful comments of the reviewers. The study is supported partly by the Ministry of Science and Technology, Taiwan under grant MOST 106-2221-E-035-096.

REFERENCES

1. R. Agrawal and R. Srikant, “Mining sequential patterns,” in *Proceedings of the 11th International Conference on Data Engineering*, 1995, pp. 3-14.

2. J. Ayres, J. Gehrke, T. Yiu, and J. Flannick, "Sequential pattern mining using a bitmap representation," in *Proceedings of the 8th ACMSIGKDD International Conference on Knowledge Discovery and Data Mining*, 2002, pp. 429-435.
3. J. Chen, "An updown directed acyclic graph approach for sequential pattern mining," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 22, 2010, pp. 913-928.
4. Y. Chen, J. Guo, Y. Wang, Y. Xiong, and Y. Zhu, "Incremental mining of sequential patterns using prefix tree," in *Proceedings of the 11th International Conference on Advances in Knowledge Discovery and Data Mining*, 2007, pp. 433-440.
5. H. Cheng, X. Yan, and J. Han, "IncSpan: Incremental mining of sequential patterns in large database," in *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2004, pp. 527-532.
6. J. K. Febrer-Hernández and J. H. Palancar, "Sequential pattern mining algorithms review," *Intelligent Data Analysis*, Vol. 16, 2012, pp. 451-466.
7. J. Huang, C. Y. Tseng, J. C. Ou, and M. S. Chen, "A general model for sequential pattern mining with a progressive database," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 20, 2008, pp. 1153-1167.
8. B. Kao, M. Zhang, C. L. Yip, D. W. Cheung, and U. M. Fayyad, "Efficient algorithms for mining and incremental update of maximal frequent sequences," *Data Mining and Knowledge Discovery*, Vol. 10, 2005, pp. 87-116.
9. R. Kohavi, C. Brodley, B. Frasca, L. Mason, and Z. Zheng, "KDD-Cup 2000 organizers' report: Peeling the onion," *SIGKDD Explorations*, Vol. 2, 2000, pp. 86-98.
10. G. Lee, Y. C. Chen, and K. C. Hung, "PTree: Mining sequential patterns efficiently in multiple data streams environment," *Journal of Information Science and Engineering*, Vol. 29, 2013, pp. 1151-1169.
11. I. H. Li, J. Y. Huang, and I. Liao, "Mining sequential pattern changes," *Journal of Information Science and Engineering*, Vol. 30, 2014, pp. 973-990.
12. M. Y. Lin and S. Y. Lee, "Incremental update on sequential patterns in large databases by implicit merging and efficient counting," *Information Systems*, Vol. 29, 2004, pp. 385-404.
13. M. Y. Lin and S. Y. Lee, "Fast discovery of sequential patterns through memory indexing and database partitioning," *Journal of Information Science and Engineering*, Vol. 21, 2005, pp. 109-128.
14. M. Y. Lin, S. C. Hsueh, and C. W. Chang, "Mining closed sequential patterns with time constraints," *Journal of Information Science and Engineering*, Vol. 24, 2008, pp. 33-46.
15. C. W. Lin, T. P. Hong, W. Gan, H. Y. Chen, and S. T. Li, "Incrementally updating the discovered sequential patterns based on pre-large concept," *Intelligent Data Analysis*, Vol. 19, 2015, pp. 1071-1089.
16. J. Liu, S. Yan, Y. Wang, and J. Ren, "Incremental mining algorithm of sequential patterns based on sequence tree," *Advances in Intelligent Systems*, Vol. 138, 2012, pp. 61-67.
17. F. Masseglia, P. Poncelet, and M. Teisseire, "Incremental mining of sequential patterns in large databases," *Data and Knowledge Engineering*, Vol. 46, 2003, pp. 97-121.
18. J. Pei, J. Han, H. Pinto, Q. Chen, U. Dayal, and M. C. Hsu, "Mining sequential pat-

terns by pattern-growth: The PrefixSpan approach," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 16, 2004, pp. 215-224.

19. R. Srikant and R. Agrawal, "Mining sequential patterns: Generalizations and performance improvements," in *Proceedings of the 5th International Conference on Extending Database Technology*, 1996, pp. 3-17.
20. X. Yan, J. Han, and R. Afshar, "CloSpan: Mining closed sequential patterns in large databases," in *Proceedings of the 3rd SIAM International Conference on Data Mining*, 2003, pp. 166-177.
21. D. Yuan, K. Lee, H. Cheng, G. Krishna, Z. Li, X. Ma, Y. Zhou, and J. Han, "CISpan: Comprehensive incremental mining algorithms of closed sequential patterns for multi-versional software mining," in *Proceedings of SIAM International Conference on Data Mining*, 2008, pp. 84-95.
22. M. J. Zaki, "Efficient enumeration of frequent sequences," in *Proceedings of ACM CIKM International Conference on Information and Knowledge Management*, 1998, pp. 68-75.

Ming-Yen Lin (林明言) received the Ph.D. degree in Department of Information Engineering and Computer Science from National Chiao Tung University, Taiwan. He is currently an Associate Professor in Feng Chia University. His research interests include pattern mining, skyline queries, and big data analytics.

Sue-Chen Hsueh (薛夙珍) received the Ph.D. degree in Department of Information management from Chiao Tung University, Taiwan. She is currently an Associate Professor in Chaoyang University of Technology. Her research interests include data mining, information security, and electronic commerce.

Chih-Chen Chan (詹志勤) received his M.S. degree in Department of Information Engineering and Computer Science from Feng Chia University. He is currently a Senior Software Engineer in MStar semiconductor, Inc. His research interests include data mining and algorithm design.