Sequence Data Mining

B ANIRUDH SRINIVSAN
COE17B019

1. Introduction

Sequential pattern mining is a topic of data mining concerned with finding statistically relevant patterns between data examples where the values are delivered in a sequence. It is usually presumed that the values are discrete, and thus time series mining is closely related, but usually considered a different activity. Sequential pattern mining is a special case of structured data mining.

There are several key traditional computational problems addressed within this field. These include building efficient databases and indexes for sequence information, extracting the frequently occurring patterns, comparing sequences for similarity, and recovering missing sequence members. In general, sequence mining problems can be classified as string mining which is typically based on string processing algorithms and itemset mining which is typically based on association rule learning.

To do sequential pattern mining, a user must provide a sequence database and specify a parameter called the minimum support threshold. This parameter indicates a minimum number of sequences in which a pattern must appear to be considered frequent, and be shown to the user. For example, if a user sets the minimum support threshold to 2 sequences, the task of sequential pattern mining consists of finding all subsequences appearing in at least 2 sequences of the input database.

2. Sequence Database

In sequential database, everything is a sequence.

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

Figure 1: Example of sequence Database.

This database contains four sequences. Each sequence represents the items purchased by a customer at different times. A sequence is an ordered list of itemsets (sets of items bought together). For example, in this database, the first sequence (SID 1) indicates that a customer bought some items a and b together, then purchased an item c, then purchased items f and g together, then purchased an item g, and then finally purchased an item e.

3. Sub sequence vs. Super sequence

A sequence is an ordered list of events, denoted $< e_1,\ e_2,\ \ldots, e_l >$. Given two sequences $a = < a_1,\ a_2,\ \ldots,\ a_n > \ and\ b = < b_1,\ b_2,\ \ldots,\ b_m >$

a is called a subsequence of b, denoted as $a \subseteq b$, if there exist integers $1 \leq j_1 < j_2 < \ldots < j_n \leq$ m such that $a_1 \subseteq b_{j1}$, $a_2 \subseteq b_{j2}$, $\ldots$, $a_n \subseteq b_{jn}$.

b is a super sequence of a. Example, a = $< (ab), d >$ and b = $< (\mathbf{ab}c), (\mathbf{d}e) >$.

4. Algos

4.1 Generalized Sequential Pattern Mining(GSP)

4.1.1 Introduction

GSP algorithm (Generalized Sequential Pattern algorithm) is an algorithm used for sequence mining. The algorithms for solving sequence mining problems are mostly based on the apriori (level-wise) algorithm. One way to use the level-wise paradigm is to first discover all the frequent items in a level-wise fashion. It simply means counting the occurrences of all singleton elements in the database. Then, the transactions are filtered by removing the non-frequent items. At the end of this step, each transaction consists of only the frequent elements it originally contained. This modified database becomes an input to the GSP algorithm. This process requires one pass over the whole database.

GSP algorithm makes multiple database passes. In the first pass, all single items (1-sequences) are counted. From the frequent items, a set of candidate 2-sequences are formed, and another pass is made to identify their frequency. The frequent 2-sequences are used to generate the candidate 3-sequences, and this process is repeated until no more frequent sequences are found. There are two main steps in the algorithm.

- **Candidate Generation**
  Given the set of frequent (k-1)-frequent sequences Fk-1, the candidates for the next pass are generated by joining F(k-1) with itself. A pruning phase eliminates any sequence, at least one of whose subsequences is not frequent.

- **Support Counting**
  Normally, a hash tree–based search is employed for efficient support counting. Finally non-maximal frequent sequences are removed.

4.1.2 **Sequence mining using GSP with an example**

Example Dataset

Let the initial candidate keys be $< A >, < B >, < C >, < D >, < E >, < F >, < G >$ and the min support be 2.

Transaction Date	Customer ID	Items Purchased	Customer Sequence
1	01	A	<AB(FG)CD>
3	01	B	
7	01	FG	
9	01	C	
10	01	D	
1	02	B	<BGD>
4	02	G	
6	02	D	
1	03	B	<BFG(AB)>
5	03	F	
8	03	G	
9	03	AB	
2	04	F	<F(AB)CD>
6	04	AB	
8	04	C	
10	04	D	
3	05	A	<A(BC)GF(DE)>
4	05	BC	
7	05	G	
9	05	F	
10	05	DE	

Figure 2: Example dataset.

Step 1: Support for candidate keys

Scan the database and find the support of all candidate keys (i.e. how many times they have appeared in the dataset).

Item	Support
A	4
B	5
C	3
D	4
~~E~~	~~1~~
F	4
G	4

Figure 3: Support counting for our candidate keys.

Step 2: Pruning

Prune all the candidate keys that didnt pass the min support. That would be all keys except $< E >$.

Step 3: Next level candidate key

We use apriori, self joining the passed candidate keys with itself, to find the next level candidate keys.

In our example,

	A	B	C	D	F	G
A		(AB)	(AC)	(AD)	(AF)	(AG)
B			(BC)	(BD)	(BF)	(BG)
C				(CD)	(CF)	(CG)
D					(DF)	(DG)
F						(FG)
G						

Figure 4: Only these are considered as candidate keys.

Step 4: Prune

Again prune all the candidate keys of this level that didnt cross the minimum support.

AA	AB	AC	AD	AF	AG
<AB(FG)CD>	<AB(FG)CD>	<AB(FG)CD>	<AB(FG)CD>	<AB(FG)CD>	<AB(FG)CD>
<BGD>	<BGD>	<BGD>	<BGD>	<BGD>	<BGD>
<BFG(AB)>	<BFG(AB)>	<BFG(AB)>	<BFG(AB)>	<BFG(AB)>	<BFG(AB)>
<F(AB)CD>	<F(AB)CD>	<F(AB)CD>	<F(AB)CD>	<F(AB)CD>	<F(AB)CD>
<A(BC)GF(DE)>	<A(BC)GF(DE)>	<A(BC)GF(DE)>	<A(BC)GF(DE)>	<A(BC)GF(DE)>	<A(BC)GF(DE)>

Figure 5: Calculating the support for all candidate keys starting with A.

~~AA~~	~~0~~	AB	2	AC	3	AD	3	AF	2	AG	2
~~BA~~	~~1~~	~~BB~~	~~1~~	BC	2	BD	4	BF	3	BG	4
~~CA~~	~~0~~	~~CB~~	~~0~~	~~CC~~	~~0~~	CD	3	~~CF~~	~~1~~	~~CG~~	~~1~~
~~DA~~	~~0~~	~~DB~~	~~0~~	~~DC~~	~~0~~	~~DD~~	~~0~~	~~DF~~	~~0~~	~~DG~~	~~0~~
FA	2	FB	2	FC	2	FD	3	~~FF~~	~~0~~	~~FG~~	~~1~~
~~GA~~	~~1~~	~~GB~~	~~1~~	~~GC~~	~~1~~	GD	3	~~GF~~	~~1~~	~~GG~~	~~0~~

Figure 6: Keys that passed the min support.

AB	AC	AD	AF	AG	BC	BD	BF	BG	CD	FA	FB	FC	FD	GD	(AB)

Figure 7: List of all the keys that passed the min support.

Step 5: Create and Prune

Keep on creating the next level candidate key and prune them. Stop when non of the candidate keys pass the min support.

2-seq. (1)	2-seq. -1st	2-seq. (2)	2-seq. -Last	3-seq after join	3-seq. after prune	Support Count	3-seq. Supported
AB	B	BC	B	ABC	ABC	1	
AB	B	BD	B	ABD	ABD	2	ABD
AB	B	BF	B	ABF	ABF	2	ABF
AB	B	BG	B	ABG	ABG	2	ABG
AB	B	(AB)	B	A(AB)			
AC	C	CD	C	ACD	ACD	3	ACD
AF	F	FA	F	AFA			
AF	F	FB	F	AFB	AFB	0	
AF	F	FC	F	AFC	AFC	1	
AF	F	FD	F	AFD	AFD	2	AFD
AG	G	GD	G	AGD	AGD	2	AGD
BC	C	CD	C	BCD	BCD	2	BCD
BF	F	FA	F	BFA			
BF	F	FB	F	BFB			
BF	F	FC	F	BFC	BFC	1	
BF	F	FD	F	BFD	BFD	2	BFD
BG	G	GD	G	BGD	BGD	3	BGD
FA	A	AB	A	FAB	FAB	0	
FA	A	AC	A	FAC	FAC	1	
FA	A	AD	A	FAD	FAD	1	
FA	A	AF	A	FAF			
FA	A	AG	A	FAG			
FA	A	(AB)	A	F(AB)	F(AB)	2	F(AB)
FB	B	BC	B	FBC	FBC	1	
FB	B	BD	B	FBD	FBD	1	
FB	B	BF	B	FBF			
FB	B	BG	B	FBG			
FC	C	CD	C	FCD	FCD	2	FCD
(AB)	B	BC	B	(AB)C	(AB)C	1	
(AB)	B	BD	B	(AB)D	(AB)D	1	
(AB)	B	BF	B	(AB)F	(AB)F	0	
(AB)	B	BG	B	(AB)G	(AB)G	0	
(AB)	A	AB	A	(AB)B			

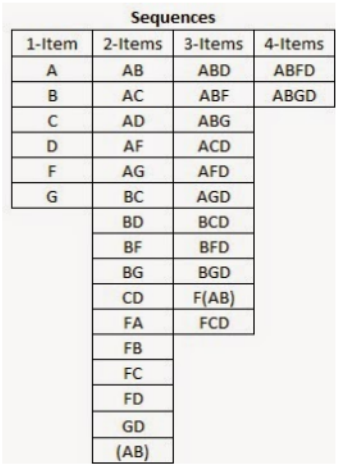
Figure 8: Creating and pruning for level 3.

3-seq. (1)	3-seq. -1st	3-seq. (2)	3-seq. -Last	4-seq. after join	4-seq. after prune	Support Count	4-seq. Supported
ABF	BF	BFD	BF	ABFD	ABFD	2	ABFD
ABG	BG	BGD	BG	ABGD	ABGD	2	ABGD

Figure 9: Creating and pruning for level 4.

Step 6: Universal set

Now the final output would be the set of all candidate keys that passed the min support at each level.



Sequences			
1-Item	2-Items	3-Items	4-Items
A	AB	ABD	ABFD
B	AC	ABF	ABGD
C	AD	ABG	
D	AF	ACD	
F	AG	AFD	
G	BC	AGD	
	BD	BCD	
	BF	BFD	
	BG	BGD	
	CD	F(AB)	
	FA	FCD	
	FB		
	FC		
	FD		
	GD		
	(AB)		

Figure 10: Final set of all the keys that passed the min support.

4.1.3 **Pseudo-Code for GSP**

```
Begin
   F1 = the set of frequent 1-sequence
   k=2,
   do while Fk-1 != Null;
       Generate candidate sets Ck (set of candidate k-sequences);
       For all input sequences s in the database D
       do
           Increment count of all a in Ck if s supports a
       End do
       Fk = {a ∈ Ck such that its frequency exceeds the threshold}
       k = k+1;
   End do
   Result = Set of all frequent sequences is the union of all Fk's
End
```

4.2 Prefix-Span Algorithm

4.2.1 Introduction

Sequential pattern mining, which discovers frequent subsequences as patterns in a sequence database, is an important data mining problem with broad applications, including the analyses of customer purchase behavior, Web access patterns, scientific experiments, disease treatments, natural disasters, DNA sequences, and so on.

The above mentioned algo faces three main probelems, namely,

- Potentially huge set of candidate sequences.
- Multiple scans of database.
- Difficulties at mining long sequential pattern.

In many applications, it is not unusual that one may encounter a large number of sequential patterns and long sequences, such as in DNA analysis or stock sequence analysis. Therefore, it is important to re-examine the sequential pattern mining problem to explore more efficient and scalable methods.

Based on our analysis, both the thrust and the bottleneck of an Apriori -based sequential pattern mining method come from its step-wise candidate sequence generation and test.

. With this motivaton, Prefix span method was developed. Its general idea is to examine only the prefix subsequences and project only their corresponding postfix subsequences into projected databases.

In each projected database, sequential patterns are grown by exploring only local frequent patterns. To further improve mining efficiency, two kinds of database projections are explored: level-by-level projection and bi-level projection.

Moreover, a main-memory-based pseudo-projection technique is developed forsaving the cost of projection and speeding up processing when the projected (sub)-database and its associated psuedo-projection processing structure can fit in main memory. Our performance study shows that bi-level projection has better performance when the database is large, and pseudo-projection speeds up the processing substantially when the projected databases can fit in memory. PrefixSpan mines the complete set of patterns and is efficient and runs considerably faster than both Apriori -based GSP algorithm and FreeSpan (Free-span is an FP-growth inspired sequence mining algorithm).

4.2.2 Some terms

1. **Prefix**

   Suppose all the items in an element are listed alphebetically. Given sequence, $\alpha = < e_1, e_2, ..., e_n >$ and $\beta = < e'_1, e'_2, ..., e'_m >$ (m < n) is called prefix of α if and only if

   (a) $e'_i = e_i \ \forall \ i \ < \ m - 1$

   (b) $e'_m \subseteq e_m$

   (c) all item in $(e_m - e'_m)$ are alphabetically after those in e'_m.

2. **Projection**

   Given a sequence α and β such that, $\beta \subseteq \alpha$. A subsequence α' of sequence α is called a projection of α w.r.t β if and only if,

   (a) α' has a prefix β

   (b) There exist no proper super sequence α'' of α' (i.e., $\alpha'' \subset \alpha'$) such that α'' is a subsequence of α and also ahs prefix β.

3. **Postfix**

   Given sequence, $\alpha' = < e_1, e_2, ..., e_n >$ be a projection of α w.r.t a prefix $\beta = < e'_1, e'_2, ..., e'_m >$ (m < n). Sequence, $\gamma = < e''_m e_{m+1}..e_n >$ is called the postfix of α w.r.t prefix β, denoted as $\gamma = \frac{\alpha}{\beta}$, where $e''_m = (e_m - e'_m)^2$. Note that if β is not a subsequence of α, both projection and postfix of α w.r.t. β is empty.

For example,

$< a >, < aa >, < a(ab) >$ and $< a(abc) >$ are prefix of sequence $< a(abc)(ac)d(cf) >$, but neither $< ab >$ nor $< a(bc) >$ is considered as prefix.

$< (abc)(ac)d(cf) >$ is the postfix of the same sequence w.r.t prefix $< a >$, $< (\_bc)(ac)d(cf) >$ is the postfix w.r.t prefix $< aa >$ and $< (\_c)(ac)d(cf) >$ is the postfix w.r.t prefix $< ab >$.

4.2.3 **Sequence mining using Prefix Spanning with an example**

Example Dataset

The min support be 2.

Sequence_id	Sequence
10	$\langle a(abc)(ac)d(cf)\rangle$
20	$\langle (ad)c(bc)(ae)\rangle$
30	$\langle (ef)(ab)(df)cb\rangle$
40	$\langle eg(af)cbc\rangle$

Figure 11: Example dataset.

Step 1: Find length-1 sequential patterns

Scan the database once to find all frequent items in sequences. Each of these frequent items is a length-1 sequential pattern. They are :

$< a >: 4, < b >: 4, < c >: 4, < d >: 3, < e >: 3, < f >: 3$

, where $< prefix >: count$.

Step 2: Divide search space

The complete set of sequential patterns can be partitioned into the following six subsets according to the six prefixes: (1) the ones having prefix $< a >$; ... ; and (6) the ones having prefix $< f >$.

Step 3: Find subsets of sequential patterns

The subsets of sequential patterns can be mined by constructing corresponding projected databases and mine each recursively. The projected databases as well as sequential patterns found in them are listed in Table 2, while the mining process is explained as follows,

First, let us find sequential patterns having prefix $< a >$. Only the sequences containing $< a >$ should be collected.ted. Moreover, in a sequence containing $< a >$, only the subsequence prefixed with the first occurrence of $< a >$, should be considered.

For example, in sequence $< (ef)(ab)(df)cb >$ only the subsequence $< (\_b)(df)cb >$ should be considered for mining sequential patterns having prefix $< a >$. Notice that $(\_b)$ means that the last element in the prefix, which is a, together with b, form one element. As another example, only the subsequence $< (abc)(ac)d(cf) >$ of sequence $< a(abc)(ac)d(cf) >$ is considered. For our example,

Sequences in S containing $< a >$ are projected w.r.t. $< a >$ to form the $< a >$ - projected database, which consists of four postfix sequences : $< (abc)(ac)d(cf) >$, $< (\_d)c(bc)(ae) >$, $< (\_b)(df)cb >$ and $< (\_f)cbc >$. By scanning $< a >$-projected database once, all the lenght-2 sequential patterns having prefix $< a >$ can be found.

They are : $< aa >: 2, < ab >: 4, < (ab) >: 2, < ac >: 4, < ad >: 2, < af >: 2$.

Recursively, all sequential having patterns prefix $< a >$ can be partitioned into 6 subsets: (1) those having prefix$< aa >$, (2) those having prefix $< ab >$, . . . , and finally, (6) those having prefix $< af >$. These subsets can be mined by constructing respective projected databases and mining each recursively.

The $< aa >$ -projected database consists of only one non-empty (postfix) subsequences having prefix $< aa >$: $< (\_bc)(ac)d(cf) >$. Since there is no hope to generate any frequent subsequence from a single sequence, the processing of $< aa >$-projected database terminates.

The $< ab >$-projected database consists of three postfix sequences: $< (\_c)(ac)d(cf) >$, $< (\_c)a >$ and $< c >$ .Recursively mining $< ab >$ -projected database returns fours equential patterns: $< (\_c) >, < (\_c)a >, < a >$ and $< c >$ (i.e. $< a(bc) >, < a(bc)a >$, $< aba >$ and $< abc >$).

The $< (ab) >$ projected sequence only consist of two sequence, $< (\_c)(ac)d(cf) >$ and $< (df)c >$, which leads to the finding of the following sequential patterns having prefix $< (ab) >$: $< c >, < d >, < f >$ and $< dc >$.

The $< ac > -$, $< ad > -$ and $< af > -$ projected databases can be constructed and recursively mined similarly. The sequential patterns found are shown in fig(12).

Similarly, we can find sequential patterns having prefix $< b >, < c >, < d >, < e >$ and $< f >$ by constructing $< b, > -, < c > -, < d > -, < e > -$ and $< f > -$ projected databases and mining them respectively. The projected databases are shown in fig(12).

Prefix	Projected (postfix) database	Sequential patterns
$\langle a \rangle$	$\langle (abc)(ac)d(cf) \rangle$, $\langle (\_d)c(bc)(ae) \rangle$, $\langle (\_b)(df)cb \rangle$, $\langle (\_f)cbc \rangle$	$\langle a \rangle$, $\langle aa \rangle$, $\langle ab \rangle$, $\langle a(bc) \rangle$, $\langle a(bc)a \rangle$, $\langle aba \rangle$, $\langle abc \rangle$, $\langle (ab) \rangle$, $\langle (ab)c \rangle$, $\langle (ab)d \rangle$, $\langle (ab)f \rangle$, $\langle (ab)dc \rangle$, $\langle ac \rangle$, $\langle aca \rangle$, $\langle acb \rangle$, $\langle acc \rangle$, $\langle ad \rangle$, $\langle adc \rangle$, $\langle af \rangle$
$\langle b \rangle$	$\langle (\_c)(ac)d(cf) \rangle$, $\langle (\_c)(ae) \rangle$, $\langle (df)cb \rangle$, $\langle c \rangle$	$\langle b \rangle$, $\langle ba \rangle$, $\langle bc \rangle$, $\langle (bc) \rangle$, $\langle (bc)a \rangle$, $\langle bd \rangle$, $\langle bdc \rangle$, $\langle bf \rangle$
$\langle c \rangle$	$\langle (ac)d(cf) \rangle$, $\langle (bc)(ae) \rangle$, $\langle b \rangle$, $\langle bc \rangle$	$\langle c \rangle$, $\langle ca \rangle$, $\langle cb \rangle$, $\langle cc \rangle$
$\langle d \rangle$	$\langle (cf) \rangle$, $\langle c(bc)(ae) \rangle$, $\langle (\_f)cb \rangle$	$\langle d \rangle$, $\langle db \rangle$, $\langle dc \rangle$, $\langle dcb \rangle$
$\langle e \rangle$	$\langle (\_f)(ab)(df)cb \rangle$, $\langle (af)cbc \rangle$	$\langle e \rangle$, $\langle ea \rangle$, $\langle eab \rangle$, $\langle eac \rangle$, $\langle eacb \rangle$, $\langle eb \rangle$, $\langle ebc \rangle$, $\langle ec \rangle$, $\langle ecb \rangle$, $\langle ef \rangle$, $\langle efb \rangle$, $\langle efc \rangle$, $\langle efcb \rangle$.
$\langle f \rangle$	$\langle (ab)(df)cb \rangle$, $\langle cbc \rangle$	$\langle f \rangle$, $\langle fb \rangle$, $\langle fbc \rangle$, $\langle fc \rangle$, $\langle fcb \rangle$

Figure 12: Projected Databases and Sequencial Patterns.

4.2.4 **Pseudo-Code for Prefix Span Algorithm**

```
Begin
Scan S|α once, find the set of frequent items b such that:
    a) b can be assembled to the last element of α to form a sequential
    pattern; or
    b) <b> can be appended to    to form a sequential pattern.

For each frequent item b, append it to α to form a sequential pattern α', and
    output α';

For each α', construct α'-projected database
S|α', and call PrefixSpan(α', l+1, S|α').
End
```