VTT INFORMATION TECHNOLOGY

RESEARCH REPORT TTE1-2001-10

MOSE

Mining Sequential Patterns

Version 1.0

21.5.2001

Jussi Ahola

Version history

Version	Date	Author(s)	Reviewer	Description
0.1	31.3.2001	Jussi Ahola	Esa Rinta-Runsala, Heikki Sikanen	Draft version
1.0	21.5.2001	Jussi Ahola		First version

Contact information

Jussi Ahola
VTT Information Technology
P.O. Box 1201, FIN-02044 VTT, Finland
Street Address: Tekniikantie 4 B, Espoo
Tel. +358 9 4561, fax +358 9 456 7024
Email: email-address@vtt.fi
Web: http://www.vtt.fi/tte/

Last modified on 21.5.2001

Contents

List of symbols and terms

constraints	The limitations associated with the sequential patterns. They may confine entire sequences, sequence elements, or the transitions from element to another.
CDIST	A method for support counting. Considers distinct occurrences of the sequential pattern.
CDIST_O	A method for support counting. The same as CDIST, but allowing overlapping of the elements.
COBJ	A method for support counting. Considers the objects whose sequence contains the sequential pattern.
cSPADE	Constrained SPADE. An algorithm for mining sequential patterns.
CWIN	A method for support counting. Considers the span-windows containing the sequential pattern.
CWINMIN	A method for support counting. Same as CWIN, but considering only the minimal windows of occurrence.
element	A part of a sequential pattern.
edge	A part of a sequential pattern indicating a transition from an element to another.
EVE	A family of algorithms for mining sequential patterns.
event	Building block of a sequence.
event-set	A set of events combined together. Also, an element of a sequential pattern.
GSP	An algorithm for mining sequential patterns.
ms	A timing constraint. The maximum difference of the first and last event of a sequential pattern.
Minepi	An algorithm for mining sequential patterns.
MSDD	An algorithm for mining sequential patterns.
ng	A timing constraint. Minimum gap between the first event of an element and the last event of a succeeding element.
node	An element of a sequential pattern.
object	The “owner” of a sequence. Each data sequence is associated with an object.
prefix	First part of a sequential pattern.
sequence	An ordered set of events or event-sets.
sequential pattern	A sequential association rule.
SPADE	An algorithm for mining sequential patterns.
SPIRIT	A family of algorithms for mining sequential patterns.
suffix	Last part of a sequential pattern.
TAG	An algorithm for mining sequential patterns.
timestamp	An attribute describing the time an event occurs.
Winepi	An algorithm for mining sequential patterns.
ws	A timing constraint. The maximum difference between the first and last event in an element.
xg	A timing constraint. Maximum gap between the last event of an element and the first event of a succeeding element.

1 Introduction

Discovering associations is one of the fundamental tasks of data mining. Its aim is to automatically seek for dependencies from vast amounts of data. The task results in so-called *association rules*, which are of form: If *A* occurs in the data then *B* occurs also. Only those rules that occur in the data frequently enough are generated.

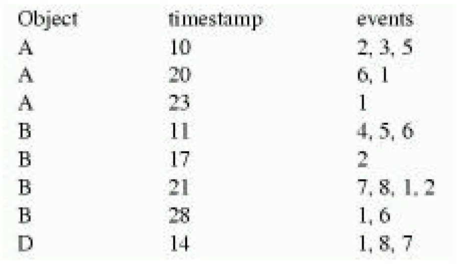
However, various information sources generate data with an inherent sequential nature, i.e., it is composed of discrete events which have a temporal/spatial ordering. This kind of data can be obtained from, e.g., telecommunications networks, electronic commerce, www-servers of Internet, and various scientific sources, like gene databases. The sequential nature of the data is totally ignored in the generation of the association rules. Thus, a part of the useful information included in the data is discarded. Thus, since the mid 90's the interest in discovering also the sequential associations in the data has arisen among the data mining community.

The sequential associations or *sequential patterns* can be presented in the form: when *A* occurs, *B* occurs within some certain time. So, the difference to traditional association rules is that here the time information is included both in the rule itself and also in the mining process in the form of timing constraints. Nowadays there exist several highly efficient methods for mining these kind of patterns. The problem with them is that they assume the input data to be sequences of discrete events including only the information of the ordering, usually the time. Often, however, the events are associated with some additional attributes. The existing methods cannot take this multi-dimensionality of the data into account and so they lose the additional information it involves. Furthermore, the methods are designed for some specific problem, and are not, as such, applicable to different types of sequential data.

In this report, a general formulation of the sequential patterns is introduced as it is presented in [1]. By using this approach the last problem of the existing algorithms can be tackled. A survey of the existing algorithm is then done. Three algorithms are presented in detail: WINEPI [2] and GSP [4] as they form the basis of the algorithms, and cSPADE [6] since it seems to be the most promising method proposed for the problem yet. Also the other relevant approaches are shortly introduced. Lastly, the extension of the patterns into the multi-dimensional is considered. Some ideas of handling the problem are given and also the features of the existing algorithms supporting multi-dimensionality are studied.

2 Sequential patterns

There exist a variety of data sources, which generate vast amounts of data, with inherent sequential nature. Such data are collected from, e.g., scientific experiments, telecommunications networks, web server logs, and transactions of supermarkets. In general the sequence data is characterised by three columns: *object*, *timestamp*, and *events*. Hence, the corresponding input records consist of occurrences of events on an object at a particular time, as illustrated in Figure 1. Depending on the data and problem in hand, various definitions of objects and events can be used. For example, object can be a customer in a bookstore and events are the book s/he bought or object can be a day and the events the switch-alarm pair of the telecommunications network.



Object	timestamp	events
A	10	2, 3, 5
A	20	6, 1
A	23	1
B	11	4, 5, 6
B	17	2
B	21	7, 8, 1, 2
B	28	1, 6
D	14	1, 8, 7

Figure 1. An example of sequential data.

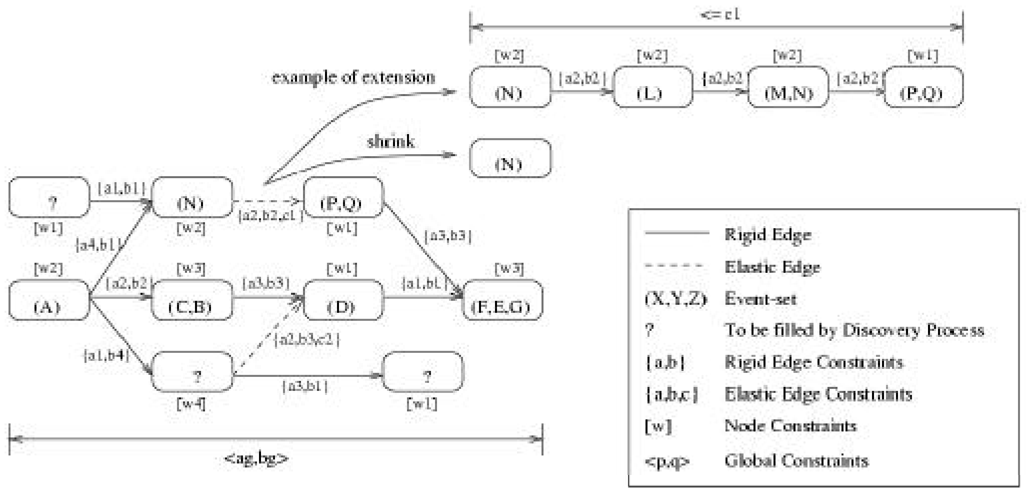
The major task associated with this kind of data is to discover sequential relationships or patterns present in these data. This can be very useful for several purposes, e.g., prediction of future events. Over the years, several different approaches have been proposed to tackle the problem. Unfortunately, the problems they assess, and thus also the resulting solutions, are very much problem dependent and often are not suitable for other types of sequential data. However, in [1] a unified formulation of sequential patterns is presented, which is an extension of the earlier approaches. Due to its generality, the formulation is also used in this report. In the following, the main aspects of the approach are considered. *If not otherwise noted, the all the concepts, examples and figures presented in the section are adopted from* [1].

The process of discovering sequential patterns involves two main issues*: the structure of the patterns in terms of its representation and constraints* and *the method by which a pattern's strength is computed*.

2.1 Structure of the sequential patterns

Most generally structure of the sequential patterns can be presented as directed acyclic graphs (DAG), see Figure 2. A graph consists of *nodes* and *directed edges*, where the former would represent elements, i.e., events or event-sets and the latter the order of their occurrences. The edges can be either elastic or rigid. During the discovery process the former can be extended into multiple edges by adding nodes dynamically in succession or shrunk by collapsing one of its incidence nodes onto the other, whereas the latter cannot be changed.

Some nodes are associated with an event-set before the discovery process, while others would be associated with different possible event-sets during it. The formers are *called event constraints*. Furthermore, *edge constraints* are called a set of numbers associated with each edge. These indicate the allowed separation between the occurrences of events of its incident nodes, e.g., minimum timegap between the occurrence of the last event in a node and the first event in the succeeding node. The nodes can also be associated with a set of numbers, *node constraints*, which determine when a set of events can be associated with a same node, e.g., the maximum timegap between the events in it. In the case of the elastic edges, the nodes of the extension share the constraints of the starting node. Similarly, the edges have the same constraints than the original edge. Furthermore, they may have an additional constraint limiting the duration of the extended edge. Beyond the edge and node constraints, there may exist *global constraints*, which are imposed on the entire discovered pattern. For example, they can limit its overall duration.



<= c1
[w2] (N) {a2,b2} [w2] (L) {a2,b2} [w2] (M,N) {a2,b2} [w1] (P,Q)
example of extension
shrink
(N)
? {a1,b1} (N) {a2,b2,c1} (P,Q)
[w1] [w2] [w1]
{a4,b1} {a3,b3}
[w2] (A) {a2,b2} [w3] (C,B) {a3,b3} [w1] (D) {a1,b1} [w3] (F,E,G)
{a2,b3,c2}
{a1,b4} ? {a3,b1} ?
[w4] [w1]
<ag,bg>
Rigid Edge
Elastic Edge
(X,Y,Z) Event-set
? To be filled by Discovery Process
{a,b} Rigid Edge Constraints
{a,b,c} Elastic Edge Constraints
[w] Node Constraints
<p,q> Global Constraints

Figure 2. General formulation of sequential relationships as a DAG.

For simplicity, without the loss of generality the multipath presentation of Figure 2 can be broken into its constituent single path sequential relationships by enumerating all the paths between the start and end nodes. Such single path relationships with all their event, node, edge, and global constraints, and two types of edges are referred to as *the universal sequential patterns*. Their formulation is given in Figure 3. By fixing the occurrence of certain events beforehand, i.e., using event constraints, the discovery algorithm can be optimised to reduce the search space. However, for the process to be meaningful at least one node should have its content unspecified. The node, edge, and global constraints of Figure 3 are called *timing constraints* and they translate into various input parameters, which govern the times at which the events in a sequence can occur:

- *ms* : **Maximum Span**, the maximum allowed time difference between the latest and earliest occurrences of events in the entire sequence.
- *ws* : **Event-set Window Size**, the maximum allowed time difference between the latest and earliest occurrences of events in any event-set.

- *xg* : **Maximum Gap**, the maximum allowed time difference between the latest occurrence of an event in an event-set and the earliest occurrence of an event in its immediately preceding event-set.
- *ng* : **Minimum Gap**, the minimum required time difference between the earliest occurrence of an event in an event-set and the latest occurrence of an event in its immediately preceding event-set.

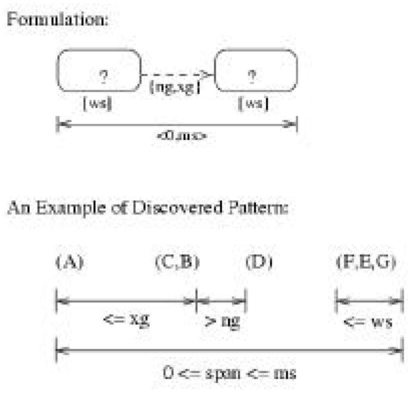


Figure 3. A universal formulation of sequential patterns.

Also an example of a discovered pattern is shown in Figure 3. It is represented as a sequence of the form <(*A*)(*C*,*B*)(*D*)(*F*,*E*,*G*)>, which denotes that the occurrence of an event *A* is followed by an event-set (*C*,*B*) succeeded by an event *D* ending with an event-set (*F*,*E*,*G*). The example illustrates the essence of the timing constraints as well.

2.2 Computation of pattern's strength

Pattern's strength is usually based on how often it occurs in the given data. If it occurs frequently enough, the pattern is said to be interesting. The two issues involved here are *how the sequence occurrence is counted* and *how many occurrences is enough*. Both issues depend on the specific application domain for which the approach is developed.

There are five different ways of counting the sequence occurrence and they can be divided into three conceptual groups. First group just looks for an occurrence of a given sequence for an object. The second group is based on counting the windows in which the given sequence occurs. Third group is based on counting the distinct occurrences of a sequence. The difference between the methods is illustrated in Figure 4.

- **COBJ** One occurrence per object.
  The count here indicates the number of objects in which the sequence appears.

- **CWIN** One occurrence per span-window.
  Span-window is defined as a window of duration equal to span (*ms*). Consecutive span-windows have one time unit's difference in their respective start and end times. They move across the entire time duration of each object and the counts for all the objects are added up.

- **CWINMIN** Number of minimal windows of occurrence.

A minimal window of occurrence is the smallest window in which the sequence occurs given the timing constraints. This definition can be considered as a restricted version of CWIN, because its effect is to shrink and collapse some of the windows that are counted by CWIN. All the minimal windows of occurrence are counted for each object and then added up over all objects.

- **CDIST_O** Distinct occurrences with possibility of event-timestamp overlap.
  A distinct occurrence of a sequence is defined to be the set of event-timestamp pairs that satisfy the specified timing constraints, such that there has to be at least one new event-timestamp pair different from the previously counted occurrences. The number of occurrences counted using this method depends on the direction in which an object's timeline is scanned (usually in the direction of increasing timestamps). All distinct occurrences are added over all objects.

- **CDIST** Distinct occurrences with no event-timestamp overlap allowed.
  This methods is similar to CDIST_O, but with overlapping disallowed. So, effectively when an event-timestamp pair is considered for counting some occurrence of a sequence, it is flagged off and is never again considered for counting occurrences of that particular sequence for that particular object.

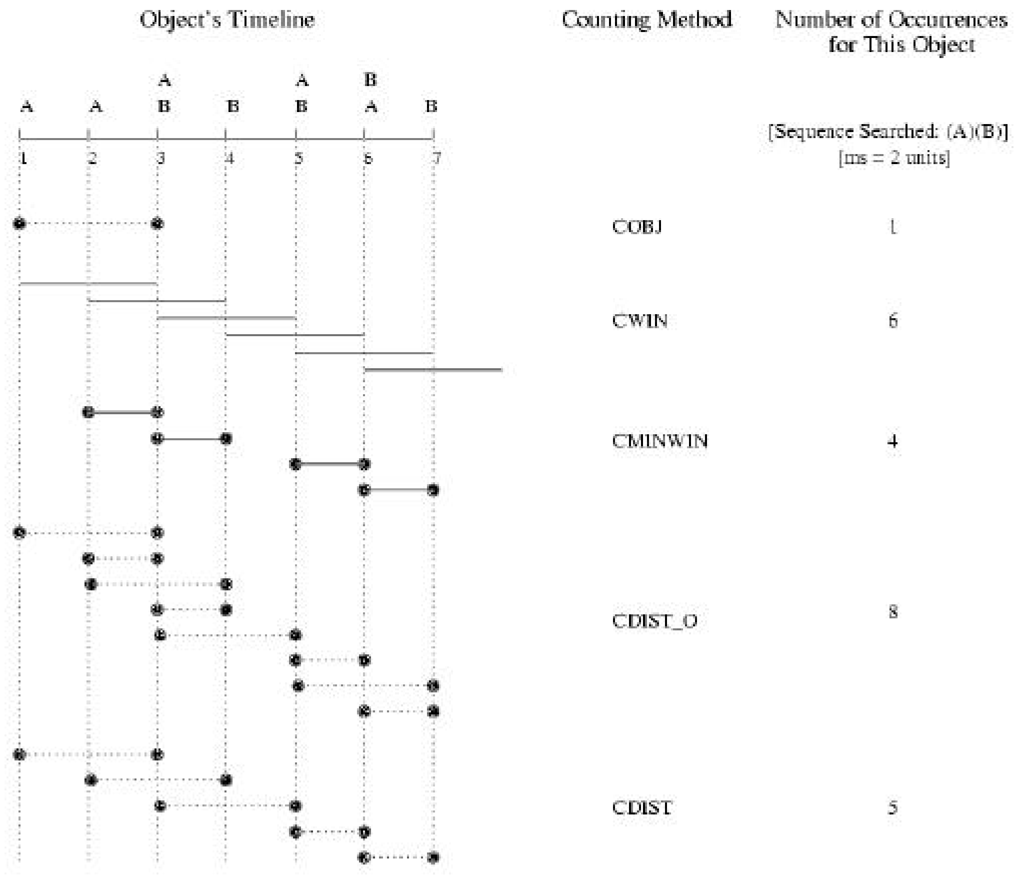


Figure 4. An example of the differences between the counting methods.

How many occurrences is enough is usually determined by the threshold of *support* which in effect is the limit determing whether the sequence is frequent enough not to be filtered out. Since sequence cannot be frequent, if any of its subsequences is not frequent, using the support threshold helps to reduce the complexity of the sequence mining algorithm used. The support can be determined as an absolute count or a percentage with respect to some basis. In the case of the latter approach, the basis depends on the counting method. For COBJ, the basis is the total number of objects in the data. For methods CWIN and CMINWIN, the basis is the sum of total number of span-windows possible in all objects. For methods CDIST and CDIST_O, the basis is the maximum number of all possible distinct occurrences of a sequence over all objects, which is the number of all distinct timestamps present in the data for each object.

The meaningfulness of the sequential patterns or rules can be evaluated using several measures of interestingness. The rule *R* can be considered to consist of an event or event-set *s* which is preceded by some set of events or event-sets *S*. Then, the *confidence* of *R* tells, what is the probability of occurrence of *s* after the occurrence of *S*. Next, the *significance* of *R* is defined as confidence of *R* per support of *s*. Thus, a high significance implies that the occurrence of *S* will most probably be followed by the occurrence of *s*. Furthermore, the *coverage* of *R* tells about the fraction of times *R* occurs with respect to the number of times *s* occurs. In the end, these measures of interestingness can be used to an efficient ordering of or filtering the discovered rules.

With the universal formulation of the sequential patterns, it is easy to try out multiple combinations of constraints and counting methods, and specify different structures of the relationships in order to develop and test different hypotheses about the sequential relationships. Hence, it can be used in different application domains and tasks.

- Discovering sequential relationships between different telecommunication switches and alarms triggering.
- Analysing data from scientific experiments conducted over a period of time.
- Discovering relationships between stock market events.
- Analysing medical records of patients for temporal patterns between diagnosis, symptoms, examination results, and treatment etc.
- Discovering patterns among different socio-economic events.

3 Mining algorithms

The approaches proposed in literature for mining sequential patterns do not, in general, utilise the universal formulation of patterns. Rather, the algorithms are designed to solve specific application related research problems, which also dictate what kind of patterns are discovered. In the following, some of the algorithms suitable for variety of problems are presented.

The terminology and notation used in the following relies on the universal formulation of the sequential patterns. So, the output of the mining process is a set of *frequent sequences*, or *sequential patterns*, denoted as *L*. Then, a *sequence*, or *pattern*, *s* denoted by $<s_1s_2s_3...s_n>$, consists of *elements* s_i, which are either *events* or *event-sets*. Alternately, the sequence can be presented as <*PS*>, where *P* is the prefix and *S* suffix of the sequence. The *length* of a sequence *s*, denoted by $|s| = k$ refers to the number of the events it contains.

So, considering the example sequence of the Figure 3 <(*A*)(*C*,*B*)(*D*)(*F*,*E*,*G*)>: Its length is seven and it is of the form $s = <s_1s_2s_3s_4>$, where s_1 consist of event *A*, s_2 of event-set (*C*,*B*), s_3 of event *D*, and s_4 of event-set (*F*,*E*,*G*). The prefix could, for instance, be (*A*)(*C*), and then the corresponding suffixes (*B*)(*D*)(*F*,*E*,*G*).

Again, unless otherwise noted, the concepts, examples and figures presented in the following are adopted from the publications referred to in the text.

3.1 WINEPI

WINEPI [3] is an algorithm, or set of algorithms, designed originally for discovering frequent sequences from a telecommunication network alarm log, which consist of a single, long sequence of alarms, or events [2]. However, the algorithms can be applied to any sequence data consisting of a single sequence, or even multiple sequences with some minor modifications.

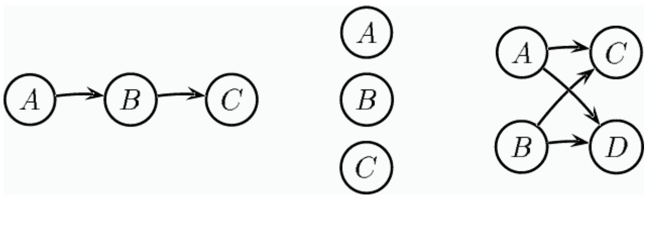


Figure 5. From left: serial sequence, parallel sequenceand the combination of these two.

The sequences can be either serial or parallel, where the former requires temporal order of events while the latter does not constrain their relative order. Also composites of these are considered, see Figure 5. Furthermore, the events of the sequences must be close enough to each other. In practise this is determined by using a time window, which is slid over the input data and only the occurrences of the sequences within the window are concerned.

The support for a sequence is determined by counting the number of windows in which it occurs.

So, referring to the universal formulation of the sequential patterns, the algorithm finds all sequences that satisfy the global time constraint *ms*, and whose support exceed a user-specific minimum *min_sup* counted with the CWIN method. The pseudo code of the main algorithm of WINEPI is presented in Figure 6.

```
1. $L_1$ = set of frequent events;
2. for ( $k$ = 2; $L_{k-1} \neq \emptyset$; $k$++)
3.   $C_k$ = New candidates of length $k$ generated from $L_{k-1}$;
4.   Compute $L_k$ , the set of frequent item-sets of length $k$;
5. end
```

Figure 6. The main algorithm of WINEPI.

As can be seen, the algorithm makes multiple passes over the data. The first pass (the first row of the algorithm above) determines the support for all individual events. In other words, for each event presented in the input data, the number of windows containing the event is counted. This results in a set of frequent events, or 1-element long sequences, which in denoted by L_1.

Each subsequent pass *k* (steps two to five of the pseudo code) starts with generating *k*-event-long candidate sequences C_k from the seed set L_{k-1}, the frequent sequences of length *k-1* found in the previous pass. This approach is motivated by the fact, that for a sequence to be frequent all of its subsequences must also be frequent. Then, the support for the candidates is counted, resulting in L_k, the set of *k*-event long frequent sequences, which again is used as a seed for the next pass. So, for instance, the second pass starts with using the frequent events L_1 as a seed, which is used to generate all potential sequences consisting of two events C_2. As their support is defined, the sequences whose support is large enough are included in L_2, which is then used as a seed for the third pass etc.

The algorithm terminates when there are no frequent sequences at the end of a pass, or when there are no candidates generated. In the following, a more detailed view of candidate generation and counting is taken.

Candidate generation

The sequences are presented as arrays of events and they are sorted into ascending order in L_{k-1}. /* */. Since the sequences are sorted, the ones sharing the same first event are consecutive in L_{k-1}. Furthermore, a group of consecutive sequences of size *k*-1 that share the first *k*-2 events is called a *block*. Potential candidates can be identified by first creating all combinations of two sequences in the same block and then pruning out the non-frequent candidates. The pseudo code for candidate generation of parallel sequences is presented in Figure 7. About the notation, the j^{th} event of the i^{th} sequence of L_{k-1} is denoted by $L_{k-1}[i][j]$ and for each sequence $L_{k-1}[i]$ the address of the first sequence of the block is stored in the array L_{k-1}.*block_start* to enable efficient identification of the blocks.

```
1. $C_k$ = ∅;
2. l = 0;
   /* for one-event-long sequences the block size is one */
3. if k == 1
4.   for ( h = 1; h ≤ |$L_k$|; h++)
5.     $L_k$.block_start[h] = 1;
     /* go through all sequences in $L_{k-1}$ */
6. for ( i = 1; i ≤ |$L_{k-1}$|; i++)
7.   current_block_start = l+1;
     /* go through all distictive blocks in $L_{k-1}$ */
8.   for ( j = i; $L_{k-1}$.block_start[j] == $L_{k-1}$.block_start[i]; j++)
       /* Since $L_{k-1}$[i] and $L_{k-1}$[j] have k-2 first events in common,
          build a potential candidate s as their combination */
9.       for ( x = 1; x ≤ k-1 ; x++)
10.        s[x] = $L_{k-1}$[i][x];
11.      s[k] = $L_{k-1}$[j][k-1];
         /* For pruning build and test subsequences s′ that do not
            contain s[x] */
12.      for ( y = 1; y < k-1 ; y++)
13.        for ( x = 1; x < y; x++)
14.          s′[x] = s[x];
15.        for ( x = y; x ≤ k-1; x++)
16.          s′[x] = s[x+1];
17.        if s′ is not in $F_{k-1}$
18.          goto row 8; /* continue with the next j */
         /* All subsequences are in $F_{k-1}$, store s as candidate */
19.      l++;
20.      $C_k$[l] = s;
21.      $C_k$.block_start[l] = current_block_start;
22.  output $C_k$
```

Figure 7. The algorithm for generating parallel candidate sequences.

For example, assuming that the frequent parallel sequences of length two are <(1,6)>, <(1,7)>, <(1,8)> and <(7,8)>. This situation corresponds, in fact, to a sequence obtained by assimilating the objects of the database of Figure 1 and using fairly low support and narrow window in the counting phase. Furthermore, since the first three sequences have the same prefix, they form the first block and the last sequence alone belongs to the second block. This situation is presented in the first two columns of Figure 8. The generation of candidates of length three starts from the first sequence, <(1,6)>, which is augmented with the last event of all the sequences in the first block (including the first sequence itself). The resulting sequences are <(1,6,6)>, <(1,6,7)>, and <(1,6,8)>. In the pruning all of the candidates can be dropped of since their subsequences <(6,6)>, <(6,7)>, and <(6,8)> are not frequent. Then the second sequence, <(1,7)> is augmented with the last events of the remaining sequences of the first block and after pruning the sequence <(1,7,8)> is included in C_3. The last sequence in the first block is then augmented with the last event of itself and the resulting sequence <(1,8,8)> is rejected in the pruning phase because its subsequence <(8,8)> is not frequent. Then the second block is processed in a similar manner. Since it consists of only one sequence, the self-augmentation is presented and again the resulting candidate can be rejected due to its subsequence <(8,8)> not being frequent. The candidate generation process is illustrated in the two last columns of the Figure 8.

Frequent parallel sequences of length two (L_2)	L_2.*block_start*	Candidates of length 3 (C_3) after join	Candidates of length 3 (C_3) after pruning
<(1,6)>	1	<(1,6,6)>, <(1,6,7)>, <(1,6,8)>	-
<(1,7)>	1	<(1,7,7)>, <(1,7,8)>	<(1,7,8)>
<(1,8)>	1	<(1,8,8)>	-
<(7,8)>	4	<(7,8,8)>	-

Figure 8. Example of the generation of parallel candidate sequences.

The algorithm generates serial sequences, if the line 8 is replaced with:

```
8.  for ( j = $L_{k-1}$.block_start[i];
              $L_{k-1}$.block_start[j] == $L_{k-1}$.block_start[i]; j++)
```

Counting candidates

The candidate counting takes advantage of the fact that consecutive windows resemble each other. The counting algorithms make incremental updates in the data structures used.

The algorithms start by considering the empty window just before the input sequence, and they end after considering the empty window just after the sequence. This way the incremental methods need no other special actions at the beginning or end. When computing the frequency of sequences, only the windows correctly on the input sequence are considered.

Parallel sequences

The main ideas of the parallel sequences counting are the following. For each candidate parallel sequence *s* there is a counter *s.event_count* that indicates how many events of *s* are present in the window. When the counter becomes equal to the length of *s*, indicating that s_p is entirely included in the window, the starting time of the window is stored in *s.inwindow*. When *s.event_count* again decreases, indicating that *s* no longer is entirely in the window, *s.freq_count* is increased by the number of windows where *s* remained entirely in the window, i.e., the time difference between the starting time of the window at issue and *s.inwindow*. At the end, *s.freq_count* contains the total number of windows where *s* occurs.

To access candidates efficiently, they are indexed by the number of events of each type that they contain: all sequences that contain exactly *a* times the event *X* are in the list *contains*(*X*,*a*). When the window is shifted and its contents changed, the sequences that are affected are updated. If, for instance, there is previously one *X* in the window and a second one comes in, all sequences in the list *contains*(*X*,2) are updated with the information that both events of type *X* they are expecting are now present. The pseudo code for counting the frequent parallel sequences is presented in Figure 9.

```
   /* Initialization */

1. for each s in C
2.   for each X in s
3.        X.count = 0;
4.        for ( i = 1; i ≤ |s|; i++ )
5.             contains(X,i) = ∅;
6. for each s in C
7.   for each X in s
8.        a = number of events of type X in s;
9.        contains(A,a) = contains(X,a)∪{s};
10.       s.event_count = 0;
11.       s.freq_count = 0;
   /* Regognition */
12.       for (start = $T_s$-win+1; $T_s$ ≤ $T_e$; start++ )
   /* Bring in new events to the window */
13.            for all events (X,t) in s such that t=start+win-1
14.                 X.count++;
15.                 for each s∈contains(X,X.count)
16.                      s.event_count += X.count;
17.                      if s.event_count = |s|
18.                           s.inwindow = start;
          /* Drop out old events from the window */
19.            for all events (X,t) in s such that t = start-1
20.                 for each s∈contains(X,X.count)
21.                      if s.event_count = |s|
22.                           s.freq_count += start – s.inwindow;
23.                      s.event_count -= X.count;
24.                 X.count--;
   /* Output */
25.       for all sequences s in C
26.            if s.freq_count/($T_e$-$T_s$+win-1) ≥ min_sup
27.                 output s;
```

Figure 9. Parallel candidate sequences counting algorithm.

Serial sequences

Serial candidate sequences are recognised by using state automata that accept the candidate sequences and ignore all other input. The idea is that there is an automaton for each serial sequence *s* and several instances of each automaton may exist at the same time, so that the active states reflect the prefixes of *s* occurring in the window. A new instance of the automaton for a sequence *s* is initialised every time the first event of *s* comes into the window. The automaton is removed when the same event leaves the window. When an automaton reaches its accepting state, indicating the sequence occurs entirely in the window, provided that there are no other automata for *s* in that state already, the starting time of the window is stored in *s.inwindow*. When an automaton in the accepting state is removed, and there are no other automata for *s* in the accepting state, the counter *s.freq_count* is incremented by the number of windows where *s* remained entirely in the window.

```
/* Initialization */
1.  for each s in C
2.     for ( i = 1; i ≤ |s|; i++ )
3.        s.initialized = 0;
4.        waits(s[i]) = ∅;
5.  for each s in C
6.     waits(s[1]) = waits(s[1])∪{(s,1)};;
7.     s.freq_count = 0;
8.  for ( t = $T_s$ – win; t<$T_s$; t++)
9.     beginsat(t) = ∅;
   /* Regognition */
10.    for (start = $T_s$-win+1; start ≤ $T_e$; start++ )
   /* Bring in new events to the window */
11.       beginsat(start+win-1) = ∅;
12.       transitions = ∅;
13.       for all events (X,t) in s such that t = start+win-1
14.          for all (s,j)∈waits(X)
15.             if j=|s| and s.initialize[j] = 0
16.                s.inwindow = start;
17.             if j = 1
18.                transitions = transitions∪{(s,1,start+win-1)};
19.             else
20.                transitions  =  transitions∪{(s,j,s.initialized[j-
   1])};
21.                beginsat(s.initialized[j-1] /= {(s,1,j-1)};
22.                s.initialized[j-1] = 0;
23.                waits(X) /= {(s,j)};
24.       for all (s,j,t)∈transitions
25.          s.initialized[j] = t;
26.          beginsat(t) ∪= {(s,j)};
27.          if j < |s|
28.             waits(s[j+1]) ∪= {(s,j+1)};
      /* Drop out old events from the window */
29.       for all {(s,l}∈beginsat(start-1)
30.          if l = |s|
31.             s.freq_count += start-s.inwindow;
32.          else
33.             waits(s[l+1]) \= {(s,l+1)};
34.          s.initialized[l] = 0;
    /* Output */
35.    for all sequences s in C
36.       if s.freq_count / ($T_e$-$T_s$+win-1) ≥ min_sup
```

Figure 10. Serial candidate sequences counting algorithm.

It is useless to have multiple automata in the same state at the same time, but it suffices to maintain the one that reached the common state last since it will also be removed last. There are thus at most |*s*| automata for a sequence *s*. For each automaton, it is needed to know, when it should be removed. Thus, all the automata for *s* can be presented in one array of size |s|, where *s*.*initialized*[*i*] refers to latest initialisation time of an automaton that has reached its *i*th state. As mentioned before, *s* itself is represented by an array containing its events; this array can be used to label the state transitions.

For efficiently accessing and traversing the automata, they are maintained as follows. For each event type *X*, the automata that accept *X* are linked together to a list *waits*(*X*). The list contains entries of the form (*s*,*j*) meaning that sequence *s* is waiting for its *j*th event. When an event (*X*,*t*) comes into the window during a shift, the list *waits*(*X*) is traversed. If an automaton reaches a common state *i* with an another automaton, the earlier entry *s*.*initialized*[*i*] is simply overwritten.

The transitions made during one shift of the window are stored in a list of *transitions*. They are presented in the form (*s*,*j*,*t*) meaning that sequence *s* got its *j*th event, and the latest initialisation time of the prefix of length *j* is *t*. Updates regarding the old states of the automata are done immediately, but updates for the new states are done only after all transitions have been identified, in order to not overwrite any useful information. For easy removal of automata when they go out of the window, the automata initialised at time *t* are stored in a list *beginsat*(*t*). The pseudo code for counting the frequent serial sequences is presented Figure 10.

Composite sequences

The recognition of an arbitrary sequence can be reduced to the recognition of a hierarchical combination of serial and parallel sequences. The occurrence of a sequence in a window can be tested using such hierarchical structure. The occurrence of a sequence s = $<s_1 s_2 ... s_n>$, where subsequences s_i may consist of either single events or parallel sequences, is done by checking whether the subsequences occur in this order, using the algorithm for serial sequences. For checking the occurrence of s_i, a method for parallel sequences is used. However, some complications have to be taken into account. First, it is sometimes necessary to duplicate an event node in the decomposition into serial and parallel sequences. Another important aspect is that composite elements have duration unlike the original events. A practical alternative is to handle all sequences like parallel, and to check the correct partial ordering only when all events are in the window. After finding the parallel sequences checking the correct partial ordering is relatively fast.

Minimal occurrences

An alternate way of discovering the frequent sequences is a method based on their *minimal occurrences*. In this approach, only the exact occurrences of the sequences and the relationships between the occurrences are considered. A minimal occurrence of a sequence in the input event sequence is determined as having an occurrence in a window w=[t_s,t_e], but not in any of its subwindows. For each frequent sequence s the locations of their minimal occurrences are stored, resulting in the set of minimal occurrences denoted by $mo(s)$={[t_s,t_e)| [t_s,t_e) is a minimal occurrence of s}. The support for a sequence is determined as the number of its minimal occurrences |$mo(s)$|. Furthermore, the approach defines *sequence rules* of the form: $s'[w_1] \Rightarrow s[w_2]$, where *s'* is a subsequence of *s* and w_1 and w_2 are windows. The interpretation of the rule is that if *s'* has a minimal occurrence at interval $[t_s,t_e)$, which is shorter than w_1, then s occurs within interval $[t_s,t_e')$ shorter than w_2. Now the approach finds all the sequences whose support exceeds the user-specified minimum and all frequent sequence rules given *w1* and *w2*. As such, the approach is similar to the universal formulation with w_2 corresponding to *ms* and an additional constraint w_1 for subsequence length, and with CMINWIN support count.

The minimal occurrence approach is simple and efficient. Furthermore, the confidence and frequency of discovered rules with a large number of different window widths are obtained quickly, i.e., the process need not to be rerun with modified window widths. Thus, the use of minimal occurrences eliminates unnecessary repetition of the recognition work, which is especially advantageous in the case of complex patterns.

A collection of algorithms for implementing this minimal occurrence based approach is called MINEPI. It is quite similar to the WINEPI presented above, and in fact, its main algorithm and the algorithm for candidate generation can be used as such. However, the candidate counting algorithms must be changed. The minimal occurrences of a candidate sequence s are located in the following way. In the first round of the main algorithm, $mo(s)$ is computed for all sequences of length one. In the subsequent rounds the minimal occurrences of s are located by first selecting its two suitable subsequences s_1 and s_2, and then performing a temporal join on their minimal occurrences. That is, for serial sequences, the subsequences are selected so that s_1 contains all events but the last one and s_2 in turn all except the first one. The minimal occurrences of s are then defined as $mo(s)=\{[t_s,u_e) \mid t_s$ being the start time of $mo(s_1)$ and u_e the end time of $mo(s_2)$ and $[t_s,u_e)$ being minimal$\}$. For parallel sequences s_1 and s_2 contain all events but one; the omitted ones must be different. The minimal occurrences of s are now specified as $mo(s)=\{[t_s,t_e) \mid t_s$ being the earlier of the start times of $mo(s_1)$ and $mo(s_2)$ and t_e similarly the latest of their end times$\}$. The minimal occurrences of a candidate can now be found in a linear pass over the minimal occurrences of the selected subsequences. However, as a setoff for this quite efficient location of the minimal occurrences, the required data structures can be even larger than the original database, especially in the first couple of iterations.

Finally, note that MINEPI can be used to solve the task of WINEPI, since a window contains an occurrence of a sequence exactly when it contains a minimal occurrence. The frequency of s can thus be computed from $mo(s)$.

Finding the sequence rules, or the sequential patterns of interest, is quite easy from the location of the minimal occurrences of a sequence. They can be enumerated by looking at all frequent sequences s, and then looking at all its subsequences s'. The evaluation of the confidence of the rule can be done in one pass through $mo(s)$ and $mo(s')$, as follows. For each $[t_s,t_e)\in mo(s')$ with $t_e-t_s \leq w_1$, locate the minimal occurrence of $[u_s,u_e)$ of s that $t_s\leq u_s$ and $[u_s,u_e)$ is the first interval in $mo(s)$ with this property. Then check whether ue-$t_s \leq w_2$. Furthermore, it is possible to incorporate the time bounds already in the initial search of the minimal occurrences resulting in a more efficient location of the minimal occurrences but also similarly bounded rule sets.

3.2 GSP

The GSP (*G*eneralised *S*equential *P*atterns) algorithm by Srikant and Agrawal [5] is designed for transaction data, where each sequence is a list of transactions ordered by transaction-time, and each transaction is a set of items. It extends their previous work on the subject [4] by enabling specifying the maximum time difference between the earliest and latest event in an element, as well as the minimum and maximum gaps between adjacent elements of the sequential pattern. Thus, the timing constraints included in this approach are *ws*, *xg*, and *ng*. In addition, the algorithm allows using a static taxonomy (is-a hierarchy) on the items. The algorithm finds all sequences that satisfy these constraints and whose support, counted with COBJ method, is greater than the user-specific minimum. In Figure 11 the algorithm is presented as a pseudo code:

```
1. $L_1$ = set of frequent events;
2. for ( $k$ = 2; $L_{k-1} \neq \emptyset$; $k$++)
3.   begin
4.        $C_k$ = New candidates of length $k$ generated from $L_{k-1}$;
5.        for all sequences $c$ in the data set
6.             Increment the count of all candidates that are
               contained in $c$;
7.        $L_k$ = set of frequent item-sets of length $k$;
8.   end
```

Figure 11. The main GSP-algorithm.

As can be seen, the algorithm works much the same way as the WINEPI described in the previous subsection. The only difference is the way the candidates are generated and their support counted. In the following, these phases are looked into more closely.

Candidate generation

Candidates are generated in two steps:

1. *Join phase.* The candidates of length k are generated by joining L_{k-1} with L_{k-1}. Let us assume that s_i and s_j are sequences belonging to L_{k-1}. Now s_i joins with s_j if the subsequence obtained by dropping the first event of s_i is the same as the subsequence obtained by dropping the last item of s_j. The resulting candidate sequence is the sequence s_i extended with the last event of s_j. The added event becomes a separate element if it was a separate element in s_j, and part of the last element of s_i otherwise. Note, that in the case of L_1, the event of s_j should be added to s_i both as part of an event-set and as a separate event.

2. *Prune phase*. All the candidates that have *contiguous* subsequences of length k-1 whose support count is less that the minimum support are deleted from C_k. The sequence c is a contiguous subsequence of s if any of the following holds:

   a) c is derived from s by dropping an event from its first or last event-set.

   b) c is derived from s by dropping an event from any of its event-sets that have at least two elements.

   c) c is a contiguous subsequence of c', which is a contiguous subsequence of s.

   If there are no max-gap constraint, i.e., $xg = \infty$, the subsequences do not need to be contiguous, but a sequence is deleted, if it has any subsequence without minimum support.

Frequent sequences of length 2	Candidates of length 4	
	after join	after pruning
<(1,6)>	<(1,6)(1)>	<(1,6)(1)>
<(1,7)>	<(1,7,8)>	<(1,7,8)>
<(1,8)>	-	-
<(7,8)>	-	-
<(1)(1)>	<(1)(1,6)>, <(1)(1,7)>, <(1)(18)>	-
<(2)(1)>	<(2)(1,6)>, <(2)(1,7)>, <(2)(1,8)>, <(2)(1)(1)>	<(2)(1,6)>, <(2)(1)(1)>
<(2)(6)>	<(2)(6)(1)>	<(2)(6)(1)>
<(5)(1)>	<(5)(1,6)>, <(5)(1,7)>, <(5)(1,8)>, <(5)(1)(1)>	<(5)(1,6)>, <(5)(1)(1)>
<(5)(6)>	<(5)(6)(1)>	<(5)(6)(1)>
<(6)(1)>	<(6)(1,6)>, <(6)(1,7)>, <(6)(1,8)>, <(6)(1)(1)>	<(6)(1)(1)>

Figure 12. An example of candidate generation.

The candidate generation process is illustrated in Figure 12. In the first column is shown a set of frequent sequences of length two from the database of Figure 1. The generation starts with the first sequence <(*1,6*)>. Thus, all sequences of form <(*6,X*)> or <(*6*)(*X*)>, where *X* is any event, are searched for. Sequence <(*6*)(*1*)> is found and joined with the first sequence. The corresponding candidate is <(*1,6*)(*1*)>. Similar procedure is repeated for all the sequences of the first column. For example, for the fifth sequence, <(*1)*(*1*)>, sequences of form <(*1X*)> or <(*1*)(*X*)> are searched for. <(*1,6*)>, <(*1,7*)>, and <(*1,8*)> are found and the candidates resulting in the join are <(*1)*(*1,6*)>, <(*1)*(*1,7*)>, and <(*1)*(*1,8*)>. All the three-event-long candidate sequences generated in the join phase are shown in the second column. The pruning phase begins by determining the contiguous subsequences of the first generated candidate <(*1,6*)(*1*)>. They are <(*1,6*)>, <(*1*)(*1*)>, and <(*6*)(*1*)>. Since they all are frequent, i.e., they are found in the first column, the sequence remains in the candidate set. The contiguous subsequences of the second generated candidate <(*1,7,8*)> are <(*1,7*)>, <(*1,8*)>, and <(*7,8*)>, which all are frequent so the candidate is also qualified to the candidate set. However, the third candidate (in the fifth row of the table in Figure 12) is rejected, since its contiguous subsequence <(*1*)(*6*)> is not frequent. After repeating this procedure for all remaining candidates, the final set of three-event-long candidate sequences C_3 are shown in the last column of the figure.

Counting candidates

The determining the support of the candidates is done by reading one data sequence at a time and incrementing the support count of candidates contained in the data sequence. The problem is thus: Given a set of candidate sequences *C* and a data sequence *d*, find all sequences in *C* that are subsequences of *d*. The solution consists of two phases:

a) A *hash tree* data structure is used to reduce the number of candidates that are checked for a data sequence.

b) The data sequence is transformed so that it can be efficiently determined whether the specific candidate is its subsequence.

Hash tree A hash tree consists of leaf nodes and interior nodes. The leaf nodes contain a list of sequences and the interior nodes a hash table. The root of the tree is defined to be at depth one and an interior node at depth p points to nodes at depth p+1.

Adding candidates All the candidates are stored to the hash tree one at a time. After defining the number of branches of the tree, the hash function to be used, and the threshold for number of sequences stored in a leaf, adding a candidate sequence goes as follows:

```
1. Start from the root, i.e., take the first event of the
   candidate under consideration.
2. Apply the hash function to the event and traverse the tree one
   step down following the branch defined by the hashing result.
3. While the node entered is not a leaf node, take the next event
   of the candidate under consideration and return to the previous
   step.
4. In the case of a leaf node, store the candidate in it, unless
   the number of the sequences in the leaf exceeds the threshold,
   in which case turn the leaf into a interior node and re-store
   [illegible]
```

Finding candidates Using the hash tree, the candidates contained in the data sequence can be found as follows:

1. Take the first event of the data sequence under inspection.
2. Start from the root node.
3. Apply the hash function to the event.
4. If the node which was arrived to is an interior node, apply the hash function to each event of the data sequence, whose transaction-time is between *t-ws* and *t+max(ws,xg)*, where *t* is the transaction-time of the event that was hashed on previous step and *ws* is the maximum allowed duration of an event-set.
5. In the case of a leaf node, check whether the data sequence contains the candidates stored in the leaf and increment the corresponding counter.
6. Take the next event of the data sequence under inspection and return to step 2.

Checking containment The algorithm for checking if the data sequence contains a candidate, corresponding to the step 5 of the algorithm above, alternates between two phases: *forward* and *backward*. The algorithm starts in the forward phase from the first element in the candidate sequence and switches between the phases until all the elements of the candidate are found in the data sequence.

a) Forward phase: The algorithm finds successive elements of the candidate in the data sequence as long as the difference between the end time of the element just found and the start time of the previous element is less than *xg*. If the difference is more than *xg*, the algorithm switches to the backward phase. If an element is not found, the data sequence does not contain the candidate sequence.

b) Backward phase: The algorithm backtracks and "pulls up" elements preceding the current element in the candidate sequence. It finds from the data sequence the first occurrence of the predecessor element whose start time is after *t-xg*, where *t* is the end time of the current element. The start time for the preceding element can be, in fact, after the end time of the current element. Furthermore, pulling up the preceding element may necessitate pulling up also its predecessor, since the *xg* constraint between them may not hold any more. The algorithm moves backwards until either the element just pulled up and the previous element satisfy *xg* or the first element has been pulled up. The algorithm then switches to the forward phase, finding elements of the candidate in the data sequence starting from the element after the last element pulled up. If any element cannot be pulled up, the data sequence does not contain the candidate sequence.

Element occurrence Concerning the algorithm just described, finding the first occurrence of an element after time *t* can be done as follows:

1. Find for each event in the element the first occurrence in the data sequence after the time *t*. This can be made efficiently by transforming the data sequence into an array where a list of its occurence times is stored for each event in the sequence. Thus, the first occurence of a certain event can be found by traversing the corresponding list of the array.
2. If the time difference between the first and last events of the element is less than or equal to *ws*, the element is found.
3. Otherwise, set *t* to the end time minus *ws* and return to the first step.

3.3 cSPADE

The cSPADE (*c*onstrained *S*equential *PA*ttern *D*iscovery using *E*quivalence classes) algorithm [7] fits for pretty much the same problems as the GSP. However, cSPADE involves constraints that are more versatile. These include, apart from those introduced in the Sec. 3.2, length, width, and duration limitations on the sequences, event constraints, and incorporating class information. The constraints are very efficiently integrated inside the algorithm. Furthermore, they are not limited only to those mentioned above, but it is easy to incorporate additional constraints as well. Also, the cSPADE has a simple implementation and it delivers an excellent performance.

cSPADE is a straightforward extension of the earlier SPADE algorithm [6], the only difference being the involvement of constraints in the cSPADE. In the following, first the original SPADE algorithm is introduced and after that, the incorporation of the constraints is described. The key features of SPADE are:

<(1)>	
object	timestamp
A	20
A	23
B	21
B	28
D	14

<(2)>	
object	timestamp
A	10
B	17
B	21

<(3)>	
object	timestamp
A	10

<(4)>	
object	timestamp
B	11

<(5)>	
object	timestamp
A	10
B	11

<(6)>	
object	timestamp
A	20
B	11
B	28

<(7)>	
object	timestamp
B	21
D	14

<(8)>	
object	timestamp
B	21
D	14

Figure 13. The idlists of the database of Figure 1.

1. *Vertical database layout*. The database layout is transformed from *horizontal* to vertical. That is, the database is re-organised so that instead of consisting of event-timestamp pairs associated with an object, the rows of the database consist of object-timestamp pairs associated with an event. From this database format it is easy to generate for each event an *idlist*, which consist of the object-timestamp tuples of the events. The idlists generated from the database of Figure 1 are shown in Figure 12. All frequent sequences can be enumerated via simple temporal joins (or intersections) of the idlists.

2. *Equivalence classes.* Equivalence prefix classes are used for decomposing the search space into smaller pieces which can be processed independently in the main memory. Usually only three database scans are required, or just one when some pre-processed information is present.

3. *Depth-first pattern search*. The problem decomposition is decoupled from the pattern search. Depth-first search is used for enumerating the frequent sequences within each equivalence class.

```
$F_1$ = set of frequent events;
$F_2$ = set of frequent 2-event-long sequences;
for all equivalence classes $[P_i] \in F_1$ in descending order
   $E_2$ = $[P_i]$;
   for ( $k$ = 3; $E_{k-1} \neq \emptyset$; $k$++)
      for all classes $[e] \in E_{k-1}$
         $N$ = process_class($[e]$);
         if ( $N \neq \emptyset$)
            $E_k = E_k \cup N$;
         delete $[e]$;
```

Figure 14. The main SPADE algorithm.

In Figure 14 the pseudo code of the SPADE algorithm is presented, given the vertical database and the user-spesific minimum support. Thus, the algorithm starts by computation of the sets of frequent events and sequences of two events. The latter is then partitioned into a set of independent equivalence classes, which are prosessed in descending order to facilitate candidate pruning. E_k denotes the set of all equivalence classes over the frequent sequences of length k and each initial equivalence class $[P_i]$ forms a sole element of E_2. At each level of the succeeding iterative phase each class $[e]$ is processed individually, using the candidate generation rules. Current class will generate a set of new equivalence classes N, which is merged with E_k and $[e]$ can be deleted. The process stops when no new frequent classes are generated.

Computing frequent events

Given the vertical idlist database, all frequent events, or one-event-long sequences can be computed in a single scan. For each event, its idlist is read to the memory and the list is scanned, incrementing the support for each new object encountered.

Computing frequent sequences of length two

For efficiency reasons, the computation of 2-event-long sequences is done in two steps. First, the database is inverted on-the-fly back to horizontal format, similar to that of Figure 1. Then, the support for all 2-sequences is counted in a single pass over this recovered database as follows:

1. Let there be n frequent events and X and Y are two of them. Then a support array of length $n \times n$ for the sequences of the form $(X)(Y)$ and of length $[n\times(n-1)]/2$ for sequences of form (XY) is set up.
2. all the input sequences, i.e., object’s timelines are gone through one by one.
3. The containment of all possible 2-event-long candidate sequences, i.e., $(X)(Y)$, $(Y)(X)$, and (XY) for events X and Y, in the object’s timeline is checked. If the candidate occurs in the input sequence, the corresponding element of the respective support array is incremented.
4. The candidates whose array elements have a support count greater than user-spesific minimum are included in F_2.

Computing frequent sequences of length k (k³3)

The two sequences of length k belong to the same equivalence class if they share a common prefix of length k-1. Thus, the elements of F_1 actually belong to the a single equivalence class, [Æ], with null prefix. On the other hand, the equivalence classes of F_2 provide a natural partition of the problem into the classes $[P_i]$, where $P_i \in F_1$. Each class can be processed independently, since it produces all frequent sequences with the prefix P_i. Thus, the large search space is partitioned into small, manageable chunks that can be processed recursively in the main memory. In the general case, however, this requires suitable memory management techniques.

So, the first thing to do is to determine the candidate elements of the last equivalence class $[P_n]$ from F_2, that is all two-event-long frequent sequences that start with P_n. These elements can be of two kind: an event-set of the form (P_nX) or a sequence of the form $(P_n)(X)$, where X is any frequent event. Then, the idlists of the corresponding elements are constructed by performing a temporal join on the idlists of P_n and X so that in former (event-set) case, the identical entries of the idlists are added in the idlist of the resulting sequence. Then again, for the latter (sequence) case, only the entries of the idlist of the subsequent event whose timestamps are greater than some timestamp with the same object in prefix’s idlist are added in the resulting idlist. Subsequently, the class $[P_n]$ is extended for the next level by constructing new candidates in three possible steps:

1. Event-set vs. event-set: Joining (P_nX) with (P_nY) results in a new event-set (P_nXY).

2. Event-set vs. sequence: Joining (P_nX) with $(P_n)(Y)$ results in a new sequence $(P_nX)(Y)$.

3. Sequence vs. sequence: Joining $(P_n)(X)$ with $(P_n)(Y)$ results in a new event-set $(P_n)(XY)$ or either sequence $(P_n)(X)(Y)$ or sequence $(P_n)(Y)(X)$. A special case is when $(P_n)(X)$ is joined with itself in which case the only possible outcome is sequence $(P_n)(X)(X)$.

However, the join is performed only, if all the subsequences of the resulting sequence are frequent, that is, they are contained in F_2. Otherwise, the sequences can be pruned out. The result are new candidate equivalence classes (subclasses of $[P_n]$) of length three. Subsequently, the new classes are processed in a similar, recursive manner for generating, including candidate construction and pruning, new classes of increasing sequence lengths until all frequent sequences with prefix P_n are found. The process is then repeated for the remaining classes $[P_i]$ until all frequent patterns are discovered.

<(1,6)>

object	timestamp
A	20
B	28

<(1,7)>

object	timestamp
B	21
D	14

<(1,8)>

object	timestamp
B	21
D	14

<(7,8)>

object	timestamp
B	21
D	14

<(1)(1)>

object	timestamp
A	23
B	28

<(2)(1)>

object	timestamp
A	23
B	21
B	28

<(2)(6)>

object	timestamp
A	20
B	28

<(5)(1)>

object	timestamp
A	20
A	23
B	21
B	28

<(5)(6)>

object	timestamp
A	20
B	28

<(6)(1)>

object	timestamp
A	23
B	21
B	28

Figure 15. Idlists of the frequent sequences of length two.

For example, considering the example of the Figure 1, the frequent two-event-long sequences are shown in the first column of Figure 12 and their idlists in figure 14. Now, the equivalence classes $[P_i]$ are [*1*] consisting of sequences <(1,6)>, <(1,7)>, <(1,8)>, and <(1)(6)>, [*2*] consisting of <(2)(1)> and <(2)(6)>, [*5*] consisting of <(5)(1)> and <(5)(6)>, [*6*] consisting of <(6)(1)>, and [*7*] consisting of <(7,8)>. The processing starts from class [*7*], which consist of only one event-set, so no next level candidates can be generated. For class [*6*] a candidate <(6)(1)(1)> can be generated by self-join but its idlist is empty implying that the sequence is not frequent. Also for class [*5*] the last candidate construction step can be applied, resulting in the candidates <(5)(1,6)>, <(5)(1)(6)>, <(5)(6)(1)>, <(5)(1)(1)>, and <(5)(6)(6)> the second and last of which can be pruned out since their subsequences <(1)(6)> and <(6)(6)> are not frequent. After the idlist joins, it turns out that idlists of sequences <(5)(1,6)> and <(5)(1)(1)> are not empty. Thus, the only next level equivalence class, [(*5*)(*1*)], consist of these two sequences. The second and third candidate construction steps can be applied to this class resulting respectively in candidate sequences <(5)(1,6)(1)> and <(5)(1)(1)(1)>. The former can be pruned out, since its subsequence <(5)(6)(1)> is not frequent and the latter is found infrequent. So, the processing of class [*5*] ends. The remaining classes are processed in a similar manner and the discovered frequent sequences are thus <(2)(1,6)>, <(2)(1)(1)>, and <(1,7,8)>.

Incorporating Constraints

Length and width restrictions

Limiting the length and/or the width of the discovered patterns has no effect on the idlists, so it is straightforward to incorporate. It can be done straightforward by simply checking in the last for loop in sixth row of Figure 14 whether the width and length of the prefix e are smaller than the user-specific limits.

Event constraints

The vertical database layout and the use of equivalence classes enable easy incorporation of the restrictions on the events that can appear on a sequence. Including a certain event in all discovered sequences is done by examining each prefix class as it is generated and by discarding the classes not containing the desired event. Due to the self-contained nature of equivalence classes, all possible sequences containing the event will eventually be generated. Respectively, excluding a certain event is done by removing the event from the set of intial classes. Thereafter, the event will never appear in any discovered sequence. The approach can be extended to handle event-sets, or sequences, as well.

Maximum span

The maximum span (*ms*) constraint is used for limiting the overall duration of the discovered sequences. Thus, it applies to the entire pattern. In SPADE it requires only a small check while performing joins. However, due to the structure of the idlist which only stores the timestamps associated with the first event of the patterns, incorporating the time window is altogether easy. So, what should somehow be known, is the time difference of the first and the last events, or the duration, of a sequence. This can be obtained by including an additional column to the idlist. Then, the idlists consist of object-timestamp-duration triples, where duration of a single event is naturally zero. Thus the overall duration of a sequences generated by join can be determined by subtracting the timestamps of antecedent element from that of the subsequent one and adding the result to the duration of the resulting sequence. If the duration of the sequence exceeds the time window, the sequence is discarded.

Minimum gap

If it is desirable that the elements of the sequence are not very near each other, minimum gap constraint is used. It defines the minimum required time difference between adjacent elements of the pattern. The incorporation of the minimum gap is done without much trouble. Only a small check for fulfillment of the user-specific minimum is needed in the join operation.

Maximum gap

The maximum gap constraint, as can be expected, defines the maximum allowed difference between adjacent elements of a sequence. As such, constraining maximum gap of a sequence destroys the class self-containment property. Due to this, unfortunately, incorporating maximum gap requires several adjustments in the processing of classes. The first change required is, as in the case of the minimum gap, the augmentation of the temporal join method to incorporate maximum gap constraint. Therefore, a join is performed only if the difference of timestamps of the antecedent and subsequent elements is below the user-specific limit.

However, since a class is no longer self-contained, a simple self-join of the class members is not sufficient. Instead, the join can be done with the set of two-event-long sequences L_2, which is already known. To extend a sequence $(P_i)(X)$ in class $[P_i]$, it is joined with all sequences of length two in the prefix class $[X]$. Thus, for each $(X)(Y)$, or (XY), in $[X]$ the candidate $(P_i)(X)(Y)$, or $(P_i)(XY)$ is generated.

3.4 Other algorithms

There is a variety of other approaches for mining sequential patterns as well. However, they are not considered to be eminently promising with respect to the scope of the of the MOSE project. In the following, some of them are shortly introduced in the level of main ideas.

3.4.1 MSDD

MSDD [8] is an acronym for Multi-Stream Dependency Detection. It is designed for mining computer network logs, but can be generalised to any multi-dimensional synchronised discrete time series data. The algorithm finds strong dependencies between patterns in the data by performing a general-to-specific, systematic search over the space of possible dependencies. The discovered sequential rules, or dependencies, are of form <*PS*>, where the prefix and suffix are both multi-dimensional events sets enabling wildcards. The algorithm involves dynamically constraints *ms.*

MSDD accepts as input a data set of sequences, each of which comprises a list of categorical events. A search over the space of possible sequential patterns is then performed, outputting the *k* (a user-specified constant) strongest patterns. The algorithm begins with a completely general pattern in which both the prefix and suffix contain only wildcards. The candidates are then generated by adding single events to either prefix or suffix. By imposing a total ordering on the adding of events, it is ensured that each candidate is generated at most once. The co-occurrence of the candidate prefix and suffix, as well as the occurrences of other same sized prefixes and suffixes resulting are counted in the contingency table. G-statistics, a statistical measure of non-independence, is calculated for the table and used as a measure of the patterns strength. The algorithm maintains a list of *k* patterns sorted by the G-statistics. For a pattern, generated from a more general pattern, an optimistic upper bound for G-statistic can be derived. Thus, the candidate can be pruned if that measure is less than the smallest measure in the maintained pattern list. The search stops when all candidates generated have no chance at being in the list.

3.4.2 TAG

The framework considered here focused on finding sequential relationships when a rough format of the relationships, or pattern structure, is given [9]. The pattern structure used here includes event constraints and time constraints with multiple granularities, and essentially it specifies what sort of patterns user is interested in. The resulting algorithm generates candidate sequences from the given pattern structure and uses TAGs (Timed Automaton with Granularities) for recognising them from the input sequence. A TAG is essentially an automaton that recognises words, but with timing information associated with the symbols. Thus, the transitions of a TAG depend both on the input symbol read and the clock values, which are maintained by the automaton and each of which is “ticking” in terms of a specific time granularity. Heuristics are used for reducing the number of candidates generated as well as decreasing the time spent for finding the frequent candidates.

The simplest version of the algorithm first derives all candidates from the given pattern structure by assigning it with the combinations of events contained in the input data. Then, a TAG for each candidate is generated. When the input data is scanned through, at every occurrence of the starting event of a candidate the corresponding TAG is started. The support can be counted by determining the number of TAGs reaching its final state.

This naive approach can be improved by several heuristic methods. These optimisations include identifying the possible inconsistencies in the given pattern structure, reducing the length of the sequence, the number of times an automaton has to be started, and the number of different automata to be started.

3.4.3 SPIRIT

The novel idea of the SPIRIT (Sequential Pattern mIning Regular ExspressIon consTraints) algorithm is to use regular expressions as a flexible constraint specification tool [10]. It involves a generic user-specific regular expression constraint on the mined patterns, thus enabling considerably versatile and powerful restrictions. In order to push the constraining inside the mining process, in practise the algorithm uses an appropriately relaxed, that is less restrictive, version of the constraint. Then it uses a post-processing step to incorporate the original constraint. There exist several versions of the algorithm, differing in the degree to which the constraints are enforced to prune the search space of patterns during computation.

The algorithm itself is very similar to the GSP-algorithm presented in Section 3.2. So, basically it works in passes, starting from the set of frequent events, and each succeeding pass resulting in the discovery of longer patterns. In the k^{th} pass, a set of candidates sequences of length *k* is generated from a set of frequent sequences of length *k-1*, and pruned. A scan over the data is then made, during which the support of the candidates is counted, resulting in a set of frequent sequences of length *k*. There are, however, some crucial differences to the GSP-like algorithms. The algorithm implements both (relaxed) constraint and support based candidate generation and pruning on each pass. At the end, the set of frequent patterns satisfying the relaxed constraint is filtered in order to check whether the output sequences satisfy also the original user-specific constraint.

3.4.4 EVE

The approach considered here uses parallel formulation [11] of sequence mining, which enables dividing both the computational work and the memory usage required among several processors. The corresponding family of algorithms called EVE (*EVE*nt distribution) implements an extended and parallel version of the GSP algorithm. The extension being due to the use of the universal formulation sequential patterns presented in the Section 2, which influences slightly on the candidate generation and counting steps of the GSP. Furthermore, the input data is distributed and the candidate hash tree data structure replicated to all processors. There are three variations of the algorithm distributing the input data differently. EVE-S distributes the entire sequences about evenly to the different processors, and is thus suited for data consisting of a relatively large number of short sequences. EVE-R is applicable when the number of objects is smaller that the number of the processors. It splits each sequence so that the total number of events assigned to different processors is similar. EVE-C is the most complex approach, which is needed in the case of few long sequences with large span-window size. The distribution of the data similar to that of the EVE-R is combined with a serial, pipeline-like co-operation of the processors needed to perform the candidate count appropriately. An additional scheme called EVECAN can be used to distribute also the candidates, together with the event sequences, among the processors. The scheme incorporates distributed hash tree structure and rotation of either the objects or candidates in round-robin manner among the processors.

3.4.5 PrefixSpan

PrefixSpan (*Prefix*-projected *S*equential *pa*tter*n* mining) method explores prefix-projection in mining of sequential patterns [12]. It greatly reduces the efforts of candidate generation. Moreover, the prefix-projection also reduces the size of projected databases and leads to an efficient processing. Its general idea is to examine only the prefix subsequences of a pattern and project only their corresponding suffix subsequences into projected databases. In each projected database, sequential patterns are grown by exploring only local frequent patterns. To further improve the mining, two kinds of database projections are considered.

The algorithm starts by finding all frequent events in the input data. The search space is then divided by partitioning the sequential patterns into subsets having the distinct frequent events as prefixes. This results in the same number of subsets as there are frequent events, the patterns in each subset starting with corresponding event. The subsets can then be mined by constructing corresponding projected databases and mine each recursively as follows. First, the database is scanned in order to find frequent events that can be assembled to the last element or added as such to the end of the prefix to form a sequential pattern. Each such pattern is then output and a respective new prefix-projected database is constructed and explored similarly. The process ends when no new frequent sequences can be generated.

There are two optimisations of the process. The cost of constructing the projected databases can be cut by using bi-level projection, which reduces the number and the size of the projected databases. Secondly, the pseudo-projection can be used to reduce the projection costs when a projected database can be held in the main memory.

4. Multi-dimensional extension of the sequential patterns

The universal formulation of the patterns considers only objects with one-dimensional discrete events, the only attribute involved being time. However, often the situation is far more complex. A very usual situation is that the objects and/or events have several additional attributes associated with them. For example, if the events are alarms from a telecommunications network, the additional attributes may involve the priority of the alarm, the identity of switch generating the alarm etc. In the case of the market basket data, attributes such as age, sex, and residence of the customers (i.e. objects) might be known.

The multi-dimensionality can be involved in the mining process in two ways. Either handling it with some kind of pre- or post-processing or pushing it inside the mining algorithm itself. The former means in practise simply incorporating existing processing methods, e.g., clustering, before or after a traditional, one-dimensional sequence mining algorithm. The latter on the other hand requires either modification of an existing sequence mining algorithm or even designing a completely new algorithm. Many of the algorithms described in the previous section have extensions that support some specific multi-dimensional features incorporating either of the approaches.

The MINEPI algorithm has an extension where the events may have arbitrary unary and event-pairs binary conditions. Thus, the events may have an arbitrary number of attributes, but the algorithm only uses them for constraining.

The GSP-algorithm allows the events to have taxonomy. This can be considered as a multi-dimensional extension, where the attributes of an event are its ancestors in the taxonomy tree. This can be generalised to arbitrary discrete attributes which GSP handles by replacing each events with an event-set consisting of the event and all its attributes. Thus, it enables uneven number of attributes for the events. Finally, the standard GSP are then run for these extended sequences.

The cSPADE algorithm has an extension applicable for data having class information presented for the sequences. The algorithm then finds frequent sequences for all different classes. This extension can be utilised when data has additional attributes associated with the objects. Then some kind of segmentation, e.g., clustering of the objects can be performed and the result can be used as class information.

Due to the general nature of the regular expression constraints used by the SPIRIT algorithm, it can probably be used also for multi-dimensional constraining. However, the algorithm itself cannot handle multi-dimensional sequences.

The MSDD algorithm is a multi-dimensional in the sense that it can handle several sequences at the time. This can be utilised by handling the event attributes as additional sequences. Although the algorithm has an advantage of being able to handle missing attribute values, it is computationally heavy.

The aim of the MOSE project is to establish a method for mining sequences consisting of objects and/or events having arbitrary number of additional discrete or categorical attributes. This is inclined to result in discovering more detailed sequential patterns, which might be advantageous when more specific information is desired to be extracted from the data. For example, in the case of market basket data, the patterns might be presented as rules like: “When a male student customer buys an expensive physics schoolbook then within a month he buys a bargain book of scifi.”

Since there is little novelty in gluing existing methods together, the preferable approach to the project is to push the multi-dimensionality inside the mining algorithm. Furthermore, the algorithm should be able to utilise the multi-dimensionality not only in the constraining but also in counting of the frequent sequences. Thus, although the approaches presented above might be very useful in some cases, they are too specific to satisfy the general goal of the MOSE project. However, some of ideas used in them can be used in the further development. Especially the extremely good performance of cSPADE (multitude of times better than, e.g., GSP or SPIRIT) tempts to try constructing the multi-dimensional algorithm based on the approach presented in it.

5. Summary

Several approaches for mining association rules from sequential data have been proposed in recent years. Typically, the approaches are developed to a specific problem in the researchers' hands. Thus, they are limited to a certain type of data and lack the generality required for the case of diverse sequential data. This problem is tackled with a universal formulation of sequential patterns.

The formulation makes use of the pre-existing approaches proposed for the problem and provides a generalised framework for sequence mining problem. It distinguishes between the pattern's structure and its support count. The structure of a pattern is presented as a directed acyclic graph consisting of nodes and edges, where the former depict the elements of a sequence and the latter its transitions from antecedent elements to subsequent. In addition, the structure includes global, node, and edge constraints, limiting the entire patterns, their elements, and the element transitions respectively. In practise, the constraints translate into various timing constraints, which dictate the duration of the sequence, the sequence elements, and allowed time difference between consecutive elements. The support count for a pattern is divided into three conceptual groups, the first of which counts only the number of objects whose sequence contain the pattern. The second one counts the number of the span-windows in which the pattern occurs, whereas the last group is based on counting only the distinct occurrences of the pattern. Finally, an appropriate measure of interestingness, such as significance, confidence, or coverage, can be used to indicate the importance of a discovered sequential pattern.

In general, the sequence mining algorithms discover all frequent sequences from the input data, where frequent means having a support greater than a user-specific limit. The most relevant algorithms exploit the apriori postulate, which states that a sequence is frequent only if all its subsequences are frequent. In practise the algorithms make multiple passes over the data, each of which consists of generation of the candidate sequences from the frequent sequences discovered in the previous pass, and counting the support of these candidates. They start by finding all the frequent single events, and iterate until no more frequent sequences can be discovered. The major differences between the algorithms are found in the approach taken in the candidates generation and counting. In addition, there exist algorithms that are based on other principles, but they are in general quite restricted and/or inefficient.

In addition to the problem-specific design of the algorithms for mining sequential patterns, they have a second shortcoming of omitting the additional attributes, which may be associated with the objects or events of the input data sequences. Often these involve useful information that would be beneficial to take into account in the mining process. However, there are no ready-made solutions for this kind of multi-dimensional data, although some of the algorithms have features supporting it. The utilisation of the multi-dimensionality can be done either by combining a separate pre- or post-processing step with an existing algorithm or by developing a new algorithm where the handling of the attributes is built in. From a research point of view, the latter is clearly a more attractive approach, so it is regarded as the preferable one for the future work. Furthermore, since there already exists extremely powerful implementations of the one-dimensional algorithms, it probably would be profitable to base the new algorithm on the ideas presented in the most efficient ones.

References

[1] Malesh Joshi, George Karypis, and Vipin Kumar, ”A Universal Formulation of Sequential Patterns,” Technical Report No. 99-021, Department of Computer Science, University of Minnesota, 1999.

[2] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo, ”Discovering Frequent Episodes in Sequences,” Proc. of the 1st Int’l Conference on Knowledge Discovery and Data Mining, Montreal, Canada, 1995.

[3] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo, ”Discovery of Frequent Episodes in Event Sequences,” Report C-1997-15, University of Helsinki, Department of Computer Science, 1997.

[4] Rakesh Agrawal and Ramakrishnan Srikant, ”Mining Sequential Patterns,” Proc. of the 11th Int’l Conference on Data Engineering, Taipei, Taiwan, 1995.

[5] Ramakrishnan Srikant and Rakesh Agrawal, ”Mining Sequential Patterns: Generalizations and Performance Improvements,” IBM Research Report RJ 9994, 1995.

[6] Mohammed J. Zaki, ”Sequence Mining in Categorical Domains: Incorporating Constraints”, Proc. of the in 9th Int’l Conference on Information and Knowledge Management, Washington, DC, 2000.

[7] Mohammed J. Zaki, ”SPADE: An Efficient Algorithm for Mining Frequent Sequences,” Machine Learning Journal, special issue on Unsupervised Learning (Doug Fisher, ed.), Vol 42, No. 1/2, 2001.

[8] Tim Oates, Matthew D. Schmill, David Jensen, and Paul R. Cohen, ”A Family of Algorithms for Finding Temporal Structure in Data,” Proc. of the 6th Int’l Workshop on AI and Statistics, Fort Lauderdale, Florida, 1997.

[9] Claudio Bettini, X. Sean Wang, and Sushil Jajodia, ”Mining Temporal Relationships with Multiple Granularities in Time Sequences,” Data Engineering Bulletin, Vol. 21, 1998.

[10] Minos N. Garofalakis, Rajeev Rastogi, and Kyuseok Shim, ”SPIRIT: Sequential Pattern Mining with Regular Expression Constraints,” Proc. of the 25th Int’l Conference on Very Large Data Bases, Edinburgh, Scotland, 1999.

[11] Mahesh V. Joshi, george Karypis, and Vipin Kumar, ”Parallel Algorithms for Mining Sequential Associations: Issues and Challenges,” Technical Report No. 00-002, Department of Computer Science, University of Minnesota, 2000.

[12] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, and Helen Pinto, ” PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth,” in Proc. of the 17th Int’l Conference on Data Engineering, Heidelberg, Germany, 2001.