Department of Computer Science
Series of Publications C
Report C-1997-15

Discovery of frequent episodes in event sequences

Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo

University of Helsinki
Finland

Discovery of frequent episodes in event sequences

Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo

University of Helsinki, Department of Computer Science
PO Box 26, FIN-00014 University of Helsinki, Finland
Email: Heikki.Mannila, Hannu.Toivonen, Inkeri.Verkamo@cs.helsinki.fi
Report C-1997-15, February 1997, 45 pages

Abstract

Sequences of events describing the behavior and actions of users or systems can be collected in several domains. We consider the problem of discovering frequently occurring episodes in such sequences. An episode is defined to be a collection of events that occur relatively close to each other in a given partial order. Once such episodes are known, one can produce rules for describing or predicting the behavior of the sequence. We give efficient algorithms for the discovery of all frequent episodes from a given class of episodes, and present extensive experimental results. The methods are in use in telecommunication alarm management.

Computing Reviews Categories and Subject Descriptors:
H.3.1 Content Analysis and Indexing
F.2.2 Nonnumerical Algorithms and Problems
I.2.6 Learning
C.2.3 Network Operations

General Terms:
Algorithms, Experimentation

Additional Key Words and Phrases:
Knowledge Discovery, Data Mining, Event Sequences, Frequent Episodes, Sequence Analysis

1 Introduction

Most data mining and machine learning techniques are adapted towards the analysis of unordered collections of data. However, there are important application areas where the data to be analyzed consists of a sequence of events. Examples of such data are alarms in a telecommunication network, user interface actions, crimes committed by a person, occurrences of recurrent illnesses, etc. Recently, interest in knowledge discovery from sequential data has increased: see, e.g., [5, 8, 17, 19, 24].

Abstractly, such data can be viewed as a sequence of events, where each event has an associated time of occurrence. An example of an event sequence is represented in Figure 1. Here A, B, C, D, E, and F are event types, e.g., different types of alarms from a telecommunication network, or different types of user actions, and they have been marked on a time line.

One basic problem in analyzing such a sequence is to find frequent *episodes*, i.e., collections of events occurring frequently together. For example, in the sequence of Figure 1, the episode "E is followed by F" occurs several times, even when the sequence is viewed through a narrow window. Episodes, in general, are partially ordered sets of events. From the sequence in the figure one can make, for instance, the observation that whenever A and B occur (in either order), C occurs soon.

When discovering episodes in a telecommunication network alarm log, the goal is to find relationships between alarms. Such relationships can then be used in an on-line analysis of the incoming alarm stream, e.g, to better explain the problems that cause alarms, to suppress redundant alarms, and to predict severe faults.

In this paper we consider the following problem. Given a class of episodes and an input sequence of events, find all episodes that occur frequently in the event sequence. We describe the framework and formalize the know-

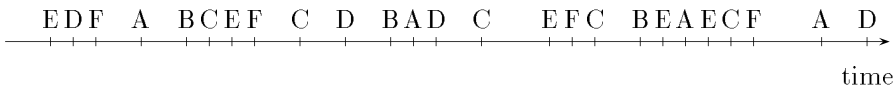

Figure 1: A sequence of events.

ledge discovery task in Section 2. Algorithms for discovering all frequent episodes are given in Section 3. Thery are based on the idea of first finding small frequent episodes, and then progressively looking for larger frequent episodes. Additionally, the algorithms use some simple pattern matching ideas to speed up the recognition of occurrences of single episodes. Section 4 outlines an alternative way of approaching the problem, based on locating minimal occurrences of episodes. Experimental results using both approaches and with various data sets are presented in Section 5. We discuss extensions and review related work in Section 6. Section 7 is a short conclusion.

2 Event sequences and episodes

Our overall goal is to analyze sequences of events, and to discover recurrent combinations of events, which we call frequent episodes. We first formulate the concept of event sequence, and then look at episodes in more detail.

2.1 Event sequences

We consider the input as a sequence of events, where each event has an associated time of occurrence. Given a set E of *event types*, an *event* is a pair (A, t), where $A \in E$ is an event type and t is an integer, the *(occurrence) time* of the event. The event type can actually contain several attributes; for simplicity we consider here the event type as a single value.

An *event sequence* s on E is a triple (s, T_s, T_e), where

$$s = \langle (A_1, t_1), (A_2, t_2), \ldots, (A_n, t_n) \rangle$$

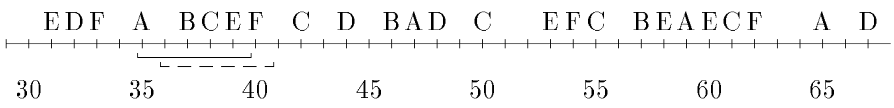

Figure 2: The example event sequence and two windows of width 5.

is an ordered sequence of events such that $A_i \in E$ for all $i = 1, \ldots, n$, and $t_i \leq t_{i+1}$ for all $i = 1, \ldots, n-1$. Further on, $T_s < T_e$ are integers, T_s is called the starting time and T_e the ending time, and $T_s \leq t_i < T_e$ for all $i = 1, \ldots, n$.

Example 1 Figure 2 presents graphically the event sequence $\mathbf{s} = (s, 29, 68)$, where

$$s = \langle (E, 31), (D, 32), (F, 33), (A, 35), (B, 37), (C, 38), \ldots, (D, 67) \rangle .$$

Observations of the event sequence have been made from time 29 to just before time 68. For each event that occurred in the time interval $[29, 68)$, the event type and the time of occurrence have been recorded. □

In the analysis of sequences we are interested in finding all frequent episodes from a class of episodes. To be considered interesting, the events of an episode must occur close enough in time. The user defines how close is close enough by giving the width of the *time window* within which the episode must occur. We define a window as a slice of an event sequence, and we then consider an event sequence as a sequence of partially overlapping windows. In addition to the width of the window, the user specifies in how many windows an episode has to occur to be considered frequent.

Formally, a *window* on event sequence $\mathbf{s} = (s, T_s, T_e)$ is an event sequence $\mathbf{w} = (w, t_s, t_e)$, where $t_s < T_e, t_e > T_s$, and w consists of those pairs (A, t) from s where $t_s \leq t < t_e$. The time span $t_e - t_s$ is called the *width* of the window $\mathbf{w}$, and it is denoted $width(\mathbf{w})$. Given an event sequence $\mathbf{s}$ and an integer win, we denote by $\mathcal{W}(\mathbf{s}, win)$ the set of all windows $\mathbf{w}$ on $\mathbf{s}$ such that $width(\mathbf{w}) = win$.

By the definition the first and last windows on a sequence extend outside the sequence, so that the first window contains only the first time point of the sequence, and the last window contains only the last time point. With this definition an event close to either end of a sequence is observed in equally many windows to an event in the middle of the sequence. Given an event sequence $\mathbf{s} = (s, T_s, T_e)$ and a window width win, the number of windows in $\mathcal{W}(\mathbf{s}, win)$ is $T_e - T_s + win - 1$.

Example 2 Figure 2 shows two windows of width 5 on the sequence $\mathbf{s}$ of the previous example. A window starting at time 35 is shown in solid line, and the immediately following window, starting at time 36, is depicted with a dashed line. The window starting at time 35 is

$$(\langle (A, 35), (B, 37), (C, 38), (E, 39) \rangle, 35, 40).$$

Note that the event $(F, 40)$ that occurred at the ending time is not in the window. The window starting at 36 is similar to this one; the difference is that the first event $(A, 35)$ is missing and there is a new event $(F, 40)$ at the end.

The set of the 43 partially overlapping windows of width 5 constitutes $\mathcal{W}(\mathbf{s}, 5)$; the first window is $(\emptyset, 25, 30)$, and the last is $(\langle (D, 67) \rangle, 67, 72)$. Event $(D, 67)$ occurs in 5 windows of width 5, as does, e.g., event $(C, 50)$. $\square$

2.2 Episodes

Informally, an episode is a partially ordered collection of events occurring together. Episodes can be described as directed acyclic graphs. Consider, for instance, episodes α, β, and γ in Figure 3. Episode α is a *serial episode*: it occurs in a sequence only if there are events of types E and F that occur

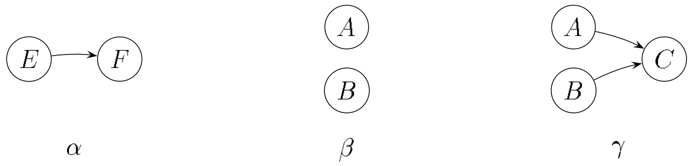

Figure 3: Episodes α, β, and γ.

in this order in the sequence. In the sequence there can be other events occurring between these two. The alarm sequence, for instance, is merged from several sources, and therefore it is useful that episodes are insensitive to intervening events. Episode β is a *parallel episode*: no constraints on the relative order of A and B are given. Episode γ is an example of non-serial and non-parallel episode: it occurs in a sequence if there are occurrences of A and B and these precede an occurrence of C; no constraints on the relative order of A and B are given. We mostly consider the discovery of serial and parallel episodes.

We now define episodes formally. An *episode* α is a triple $(V, \leq, g)$ where V is a set of nodes, $\leq$ is a partial order on V, and $g : V \to E$ is a mapping associating each node with an event type. The interpretation of an episode is that the events in $g(V)$ have to occur in the order described by $\leq$. The *size* of α, denoted $|\alpha|$, is $|V|$. Episode α is *parallel* if the partial order $\leq$ is trivial (i.e., $x \not\leq y$ for all $x, y \in V$ such that $x \neq y$). Episode α is *serial* if the relation $\leq$ is a total order (i.e., $x \leq y$ or $y \leq x$ for all $x, y \in V$). Episode α is *injective* if the mapping g is an injection, i.e., no event type occurs twice in the episode.

Example 3 Consider episode $\alpha = (V, \leq, g)$ in Figure 3. The set V contains two nodes; say x and y. The mapping g labels these nodes with the event types that are seen in the figure: $g(x) = E$ and $g(y) = F$. An event of type E is supposed to occur before an event of type F, i.e., x precedes y, and we have $x \leq y$. Episode α is injective, since it does not contain duplicate event types; in a window where α occurs there may, however, be multiple events of types E and F. $\square$

We next define when an episode is a subepisode of another; this relation is used extensively in the algorithms for discovering all frequent episodes. An episode $\beta = (V', \leq', g')$ is a *subepisode* of $\alpha = (V, \leq, g)$, denoted $\beta \preceq \alpha$, if there exists an injective mapping $f : V' \to V$ such that $g'(v) = g(f(v))$ for all $v \in V'$, and for all $v, w \in V'$ with $v \leq' w$ also $f(v) \leq f(w)$. An episode α is a *superepisode* of β if and only if $\beta \preceq \alpha$. We write $\beta \prec \alpha$ if $\beta \preceq \alpha$ and $\alpha \not\preceq \beta$.

Example 4 From Figure 3 we see that $\beta \preceq \gamma$ since β is a subgraph of γ. In terms of the definition, there is a mapping f that connects the nodes labeled A with each other and the nodes labeled B with each other, i.e., both nodes of β have (disjoint) corresponding nodes in γ. Since the nodes in episode β are not ordered, the corresponding nodes in γ do not need to be ordered, either. $\square$

Consider now what it means that an episode occurs in a sequence. The nodes of the episode need to have corresponding events in the sequence such that the event types are the same and the partial order of the episode is respected. An episode $\alpha = (V, \leq, g)$ *occurs* in an event sequence

$$\mathbf{s} = (\langle (A_1, t_1), (A_2, t_2), \ldots, (A_n, t_n) \rangle, T_s, T_e),$$

if there exists an injective mapping $h : V \to \{1, \ldots, n\}$ from nodes to events, such that $g(x) = A_{h(x)}$ for all $x \in V$, and for all $x, y \in V$ with $x \neq y$ and $x \leq y$ we have $t_{h(x)} < t_{h(y)}$.

Example 5 The window $(w, 35, 40)$ of Figure 2 contains events A, B, C, and E. Episodes β and γ of Figure 3 occur in the window, but α does not. $\square$

Algorithm 1
Input: A set E of event types, an event sequence $\mathbf{s}$ over E, a set $\mathcal{E}$ of episodes, a window width *win*, a frequency threshold *min_fr*, and a confidence threshold *min_conf*.
Output: The episode rules that hold in $\mathbf{s}$ with respect to *win*, *min_fr*, and *min_conf*.
Method:

/* Find frequent episodes (Algorithm 2): */
compute $\mathcal{F}(\mathbf{s}, win, min\_fr)$;
/* Generate rules: */
for all $\alpha \in \mathcal{F}(\mathbf{s}, win, min\_fr)$ **do**
    for all $\beta \prec \alpha$ **do**
        if $fr(\alpha)/fr(\beta) \geq min\_conf$ **then**
            output the rule $\beta \rightarrow \alpha$ and the confidence $fr(\alpha)/fr(\beta)$;

We define the *frequency* of an episode as the fraction of windows in which the episode occurs. That is, given an event sequence $\mathbf{s}$ and a window width *win*, the frequency of an episode α in $\mathbf{s}$ is

$$fr(\alpha, \mathbf{s}, win) = \frac{|\{\mathbf{w} \in \mathcal{W}(\mathbf{s}, win) \mid \alpha \text{ occurs in } \mathbf{w}\}|}{|\mathcal{W}(\mathbf{s}, win)|}.$$

Given a *frequency threshold min_fr*, α is *frequent* if $fr(\alpha, \mathbf{s}, win) \geq min\_fr$. The task we are interested in is to discover all frequent episodes from a given class $\mathcal{E}$ of episodes. The class could be, e.g., all parallel episodes or all serial episodes. We denote the collection of frequent episodes with respect to $\mathbf{s}$, *win* and *min_fr* by $\mathcal{F}(\mathbf{s}, win, min\_fr)$.

Once the frequent episodes are known, they can be used to obtain rules that describe connections between events in the given event sequence. For example, if we know that the episode β of Figure 3 occurs in 4.2 % of the windows and that the superepisode γ occurs in 4.0 % of the windows, we can estimate that after seeing a window with A and B, there is a chance of about 0.95 that C follows in the same window. Such rules show the connections between events more clearly than frequent episodes alone. Algorithm 1 shows how rules and their confidences can be computed from the frequencies of

Algorithm 2
Input: A set E of event types, an event sequence $\mathbf{s}$ over E, a set $\mathcal{E}$ of episodes, a window width *win*, and a frequency threshold *min_fr*.
Output: The collection $\mathcal{F}(\mathbf{s}, win, min\_fr)$ of frequent episodes.
Method:

compute $\mathcal{C}_1 := \{\alpha \in \mathcal{E} \mid |\alpha| = 1\}$;
$l := 1$;
while $\mathcal{C}_l \neq \emptyset$ **do**
    /* Database pass (Algorithms 4 and 5): */
    compute $\mathcal{F}_l := \{\alpha \in \mathcal{C}_l \mid fr(\alpha, \mathbf{s}, win) \geq min\_fr\}$;
    $l := l + 1$;
    /* Candidate generation (Algorithm 3): */
    compute $\mathcal{C}_l := \{\alpha \in \mathcal{E} \mid |\alpha| = l$ and for all $\beta \in \mathcal{E}$ such that $\beta \prec \alpha$ and
        $|\beta| < l$ we have $\beta \in \mathcal{F}_{|\beta|}\}$;
for all l **do** output $\mathcal{F}_l$;

---

episodes. Note that indentation is used in the algorithms to specify the extent of loops and conditional statements.

3 Algorithms

Given all frequent episodes, the rule generation is straightforward. We now concentrate on the following discovery task: given an event sequence $\mathbf{s}$, a set $\mathcal{E}$ of episodes, a window width *win*, and a frequency threshold *min_fr*, find $\mathcal{F}(\mathbf{s}, win, min\_fr)$. We give first a specification of the algorithm and then exact methods for its subtasks. We call these methods collectively the WINEPI algorithm.

3.1 Main algorithm

Algorithm 2 computes the collection $\mathcal{F}(\mathbf{s}, win, min\_fr)$ of frequent episodes from a class $\mathcal{E}$ of episodes. The algorithm performs a levelwise (breadth-first) search in the episode lattice spanned by the subepisode relation. The search starts from the most general episodes, i.e., episodes with only one event. On each level the algorithm first computes a collection of candidate episodes, and then checks their frequencies from the event sequence database. The crucial point in the candidate generation is given by the following immediate lemma.

Lemma 6 If an episode α is frequent in an event sequence $\mathbf{s}$, then all subepisodes $\beta \preceq \alpha$ are frequent.

The collection of candidates is specified to consist of episodes such that all smaller subepisodes are frequent. This criterion safely prunes from consideration episodes that can not be frequent. More detailed methods for the candidate generation and database pass phases are given in the following subsections.

3.2 Generation of candidate episodes

We present now a candidate generation method in detail. The method can be easily adapted to deal with the classes of parallel episodes, serial episodes, and injective parallel and serial episodes.

Algorithm 3 computes candidates for parallel episodes. In the algorithm, an episode $\alpha = (V, \leq, g)$ is represented as a lexicographically sorted array of event types. The array is denoted by the name of the episode and the items in the array are referred to with the square bracket notation. For example, a parallel episode α with events of types A, C, C, and F is represented as an array α with $\alpha[1] = A, \alpha[2] = C, \alpha[3] = C$, and $\alpha[4] = F$. Collections of episodes are also represented as lexicographically sorted arrays, i.e., the ith episode of a collection $\mathcal{F}$ is denoted by $\mathcal{F}[i]$.

Since the episodes and episode collections are sorted, all episodes that share the same first event types are consecutive in the episode collection. In particular, if episodes $\mathcal{F}_l[i]$ and $\mathcal{F}_l[j]$ of size l share the first $l-1$ events, then for all k with $i \leq k \leq j$ we have that $\mathcal{F}_l[k]$ shares also the same

Algorithm 3
Input: A sorted array $\mathcal{F}_l$ of frequent parallel episodes of size l.
Output: A sorted array of candidate parallel episodes of size $l+1$.
Method:

1. $\mathcal{C}_{l+1} := \emptyset$;
2. $k := 0$;
3. **if** $l = 1$ **then for** $h := 1$ **to** $|\mathcal{F}_l|$ **do** $\mathcal{F}_l.block\_start[h] := 1$;
4. **for** $i := 1$ **to** $|\mathcal{F}_l|$ **do**
5. $current\_block\_start := k + 1$;
6.   **for** $(j := i; \mathcal{F}_l.block\_start[j] = \mathcal{F}_l.block\_start[i]; j := j + 1)$ **do**
7.   /* $\mathcal{F}_l[i]$ and $\mathcal{F}_l[j]$ have $l-1$ first event types in common,
8.   build a potential candidate α as their combination: */
9.   **for** $x := 1$ **to** l **do** $\alpha[x] := \mathcal{F}_l[i][x]$;
10. $\alpha[l+1] := \mathcal{F}_l[j][l]$;
11.   /* Build and test subepisodes β that do not contain $\alpha[y]$: */
12.   **for** $y := 1$ **to** $l-1$ **do**
13.   **for** $x := 1$ **to** $y-1$ **do** $\beta[x] := \alpha[x]$;
14.   **for** $x := y$ **to** l **do** $\beta[x] := \alpha[x+1]$;
15.   **if** β is not in $\mathcal{F}_l$ **then** continue with the next j at line 6;
16.   /* All subepisodes are in $\mathcal{F}_l$, store α as candidate: */
17. $k := k + 1$;
18. $\mathcal{C}_{l+1}[k] := \alpha$;
19. $\mathcal{C}_{l+1}.block\_start[k] := current\_block\_start$;
20. output $\mathcal{C}_{l+1}$;

---

events. A maximal sequence of consecutive episodes of size l that share the first $l-1$ events is called a *block*. Potential candidates can be identified by creating all combinations of two episodes in the same block. For the efficient identification of blocks, we store in $\mathcal{F}_l.block\_start[j]$ for each episode $\mathcal{F}_l[j]$ the i such that $\mathcal{F}_l[i]$ is the first episode in the block.

Algorithm 3 can be easily modified to generate candidate serial episodes. Now the events in the array representing an episode are in the order imposed by a total order $\leq$. For instance, a serial episode β with events of types C, A, F, and C, in that order, is represented as an array β with $\beta[1] = C$, $\beta[2] = A$, $\beta[3] = F$, and $\beta[4] = C$. By replacing line 6 by

6. **for** $(j := \mathcal{F}_l.block\_start[i]; \mathcal{F}_l.block\_start[j] = \mathcal{F}_l.block\_start[i]; j := j + 1)$ **do**

Algorithm 3 generates candidates for serial episodes.

There are further options with the algorithm. If the desired episode class consists of parallel or serial injective episodes, i.e., no episode should contain any event type more than once, simply insert line

6*b*. **if** $j = i$ **then** continue with the next j at line 6;

after line 6.

The time complexity of Algorithm 3 is polynomial in the size of the collection of frequent episodes and it is independent of the length of the event sequence.

Theorem 1 Algorithm 3 (with any of the above variations) has time complexity $\mathcal{O}(l^2 \, |\mathcal{F}_l|^2 \log |\mathcal{F}_l|)$.

Proof The initialization (line 3) takes time $\mathcal{O}(|\mathcal{F}_l|)$. The outer loop (line 4) is iterated $\mathcal{O}(|\mathcal{F}_l|)$ times and the inner loop (line 6) $\mathcal{O}(|\mathcal{F}_l|)$ times. Within the loops, a potential candidate (lines 9 and 10) and $l-1$ subcandidates (lines 12 to 14) are built in time $\mathcal{O}(l+1+(l-1)l) = \mathcal{O}(l^2)$. More importantly, the $l-1$ subsets need to be searched for in the collection $\mathcal{F}_l$ (line 15). Since $\mathcal{F}_l$ is sorted, each subcandidate can be located with binary search in time $\mathcal{O}(l \log |\mathcal{F}_l|)$. The total time complexity is thus $\mathcal{O}(|\mathcal{F}_l| + |\mathcal{F}_l| \, |\mathcal{F}_l| \, (l^2 + (l-1) \, l \, \log |\mathcal{F}_l|)) = \mathcal{O}(l^2 \, |\mathcal{F}_l|^2 \log |\mathcal{F}_l|)$. □

In practical situations the time complexity is likely to be close to $\mathcal{O}(l^2 \, |\mathcal{F}_l| \, \log |\mathcal{F}_l|)$, since the blocks are typically small.

3.3 Recognizing episodes in sequences

Let us now consider the implementation of the database pass. We give algorithms which recognize episodes in sequences in an incremental fashion. For two windows $\mathbf{w} = (w, t_s, t_s + win)$ and $\mathbf{w}' = (w', t_s + 1, t_s + win + 1)$, the sequences w and w' of events are similar to each other. We take advantage of this similarity: after recognizing episodes in $\mathbf{w}$, we make incremental updates in our data structures to achieve the shift of the window to obtain $\mathbf{w}'$.

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 episodes, only the windows correctly on the input sequence are, of course, considered.

3.3.1 Parallel episodes

Algorithm 4 recognizes candidate parallel episodes in an event sequence. The main ideas of the algorithm are the following. For each candidate parallel episode α we maintain a counter $\alpha.event\_count$ that indicates how many events of α are present in the window. When $\alpha.event\_count$ becomes equal to $|\alpha|$, indicating that α is entirely included in the window, we save the starting time of the window in $\alpha.inwindow$. When $\alpha.event\_count$ decreases again, indicating that α is no longer entirely in the window, we increase the field $\alpha.freq\_count$ by the number of windows where α remained entirely in the window. At the end, $\alpha.freq\_count$ contains the total number of windows where α occurs.

To access candidates efficiently, they are indexed by the number of events of each type that they contain: all episodes that contain exactly a events of type A are in the list *contains(A,a)*. When the window is shifted and the contents of the window change, the episodes that are affected are updated. If, for instance, there is one event of type A in the window and a second one comes in, all episodes in the list *contains(A,2)* are updated with the information that both events of type A they are expecting are now present.

Algorithm 4
Input: A collection $\mathcal{C}$ of parallel episodes, an event sequence $\mathbf{s} = (s, T_s, T_e)$, a window width *win*, and a frequency threshold *min_fr*.
Output: The episodes of $\mathcal{C}$ that are frequent in $\mathbf{s}$ with respect to *win* and *min_fr*.
Method:

```
/* Initialization: */
for each α in C do
    for each A in α do
        A.count := 0;
        for i := 1 to |α| do contains(A, i) := ∅;
for each α in C do
    for each A in α do
        a := number of events of type A in α;
        contains(A, a) := contains(A, a) ∪ {α};
    α.event_count := 0;
    α.freq_count := 0;
/* Recognition: */
for start := T_s − win + 1 to T_e do
    /* Bring in new events to the window: */
    for all events (A, t) in s such that t = start + win − 1 do
        A.count := A.count + 1;
        for each α ∈ contains(A, A.count) do
            α.event_count := α.event_count + A.count;
            if α.event_count = |α| then α.inwindow := start;
    /* Drop out old events from the window: */
    for all events (A, t) in s such that t = start − 1 do
        for each α ∈ contains(A, A.count) do
            if α.event_count = |α| then
                α.freq_count := α.freq_count − α.inwindow + start;
            α.event_count := α.event_count − A.count;
        A.count := A.count − 1;
/* Output: */
for all episodes α in C do
    if α.freq_count/(T_e − T_s + win − 1) ≥ min_fr then output α;
```

3.3.2 Serial episodes

Serial candidate episodes are recognized in an event sequence by using state automata that accept the candidate episodes and ignore all other input. The idea is that there is an automaton for each serial episode α, and that there can be several instances of each automaton at the same time, so that the active states reflect the (disjoint) prefixes of α occurring in the window. Algorithm 5 implements this idea.

We initialize a new instance of the automaton for a serial episode α every time the first event of α comes into the window; the automaton is removed when the same event leaves the window. When an automaton for α reaches its accepting state, indicating that α is entirely included in the window, and if there are no other automata for α in the accepting state already, we save the starting time of the window in $\alpha.inwindow$. When an automaton in the accepting state is removed, and if there are no other automata for α in the accepting state, we increase the field α.*freq_ count* by the number of windows where α remained entirely in the window.

It is useless to have multiple automata in the same state, as they would only make the same transitions and produce the same information. It suffices to maintain the one that reached the common state last since it will be also removed last. There are thus at most $|\alpha|$ automata for an episode α. For each automaton we need to know when it should be removed. We can thus represent all the automata for α with one array of size $|\alpha|$: the value of $\alpha.initialized[i]$ is the latest initialization time of an automaton that has reached its ith state. Recall that α itself is represented by an array containing its events; this array can be used to label the state transitions.

To access and traverse the automata efficiently they are organized in the following way. For each event type $A \in E$, the automata that accept A are linked together to a list *waits(A)*. The list contains entries of the form (α, x) meaning that episode α is waiting for its xth event. When an event (A, t) enters the window during a shift, the list *waits(A)* is traversed. If an automaton reaches a common state i with another automaton, the earlier entry $\alpha.initialized[i]$ is simply overwritten.

The transitions made during one shift of the window are stored in a list *transitions*. They are represented in the form (α, x, t) meaning that episode

Algorithm 5

Input: A collection $\mathcal{C}$ of serial episodes, an event sequence $\mathbf{s} = (s, T_s, T_e)$, a window width win, and a frequency threshold min_fr.

Output: The episodes of $\mathcal{C}$ that are frequent in $\mathbf{s}$ with respect to win and min_fr.

Method:

```
/* Initialization: */
for each α in C do
    for i := 1 to |α| do
        α.initialized[i] := 0;
        waits(α[i]) := ∅;
for each α ∈ C do
    waits(α[1]) := waits(α[1]) ∪ {(α, 1)};
    α.freq_count := 0;
for t := T_s − win to T_s − 1 do beginsat(t) := ∅;
/* Recognition: */
for start := T_s − win + 1 to T_e do
    /* Bring in new events to the window: */
    beginsat(start + win − 1) := ∅;
    transitions := ∅;
    for all events (A, t) in s such that t = start + win − 1 do
        for all (α, j) ∈ waits(A) do
            if j = |α| and α.initialized[j] = 0 then α.inwindow := start;
            if j = 1 then
                transitions := transitions ∪ {(α, 1, start + win − 1)};
            else
                transitions := transitions ∪ {(α, j, α.initialized[j − 1])};
                beginsat(α.initialized[j − 1]) :=
                                beginsat(α.initialized[j − 1]) \ {(α, j − 1)};
                α.initialized[j − 1] := 0;
                waits(A) := waits(A) \ {(α, j)};
    for all (α, j, t) ∈ transitions do
        α.initialized[j] := t;
        beginsat(t) := beginsat(t) ∪ {(α, j)};
        if j < |α| then waits(α[j + 1]) := waits(α[j + 1]) ∪ {(α, j + 1)};
    /* Drop out old events from the window: */
    for all (α, l) ∈ beginsat(start − 1) do
        if l = |α| then α.freq_count := α.freq_count − α.inwindow + start;
        else waits(α[l + 1]) := waits(α[l + 1]) \ {(α, l + 1)};
        α.initialized[l] := 0;
/* Output: */
for all episodes α in C do
    if α.freq_count/(T_e − T_s + win − 1) ≥ min_fr then output α;
```

α got its xth event, and the latest initialization time of the prefix of length x 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 initialized at time t are stored in a list *beginsat(t)*.

3.3.3 Analysis of time complexity

For simplicity, suppose that the class of event types E is fixed, and assume that exactly one event takes place every time unit. Assume candidate episodes are all of size l, and let n be the length of the sequence.

Theorem 2 The time complexity of Algorithm 4 is $\mathcal{O}((n + l^2)\,|\mathcal{C}|)$.

Proof Initialization takes time $\mathcal{O}(|\mathcal{C}|\,l^2)$. Consider now the number of the operations in the innermost loops, i.e., accesses to $\alpha.event\_count$ on lines 18 and 25. In the recognition phase there are $\mathcal{O}(n)$ shifts of the window. In each shift, one new event comes into the window, and one old event leaves the window. Thus, for any episode α, $\alpha.event\_count$ is accessed at most twice during one shift. The cost of the recognition phase is thus $\mathcal{O}(n\,|\mathcal{C}|)$. $\square$

In practice the size l of episodes is very small with respect to the size n of the sequence, and the time required for the initialization can be safely neglected. For injective episodes we have the following tighter result.

Theorem 3 The time complexity of recognizing injective parallel episodes in Algorithm 4 (excluding initialization) is $\mathcal{O}(\frac{n}{win}\,|\mathcal{C}|\,l + n)$.

Proof Consider *win* successive shifts of one time unit. During such sequence of shifts, each of the $|\mathcal{C}|$ candidate episodes α can undergo at most $2l$ changes: any event type A can have $A.count$ increased to 1 and decreased to 0 at most once. This is due to the fact that after an event of type A has come into the window, $A.count \geq 1$ for the next win time units. Reading the input takes time n. $\square$

This time bound should be contrasted with the time usage of a trivial non-incremental method where the sequence is pre-processed into windows, and then frequent sets are searched for. The time requirement for recognizing $|\mathcal{C}|$ candidate sets in n windows, plus the time required to read in n windows of size win, is $\mathcal{O}(n\,|\mathcal{C}|\,l + n \cdot win)$, i.e., larger by a factor of win.

Theorem 4 The time complexity of Algorithm 5 is $\mathcal{O}(n\,|\mathcal{C}|\,l)$.

Proof The initialization takes time $\mathcal{O}(|\mathcal{C}|\,l + win)$. In the recognition phase, again, there are $\mathcal{O}(n)$ shifts, and in each shift one event comes into the window and one event leaves the window. In one shift, the effort per an episode α depends on the number of automata accessed; there are a maximum of l automata for each episode. The worst-case time complexity is thus $\mathcal{O}(|\mathcal{C}|\,l + win + n\,|\mathcal{C}|\,l) = \mathcal{O}(n\,|\mathcal{C}|\,l)$ (note that win is $\mathcal{O}(n)$). $\square$

In the worst case the input sequence consists of events of only one event type, and the candidate serial episodes consist only of events of that particular type. Every shift of the window results now in an update in every automaton. This worst-case complexity is close to the complexity of the trivial non-incremental method $\mathcal{O}(n\,|\mathcal{C}|\,l + n \cdot win)$. In practical situations, however, the time requirement is considerably smaller, and we approach the savings obtained in the case of injective parallel episodes.

Theorem 5 The time complexity of recognizing injective serial episodes in Algorithm 5 (excluding initialization) is $\mathcal{O}(n\,|\mathcal{C}|)$.

Proof Each of the $\mathcal{O}(n)$ shifts can now affect at most two automata for each episode: when an event comes into the window there can be a state

Figure 4: Recursive composition of a complex episode.

transition in at most one automaton, and at most one automaton can be removed because the initializing event goes out of the window. □

3.4 General partial orders

So far we have only discussed serial and parallel episodes. We next discuss briefly the use of other partial orders in episodes. The recognition of an arbitrary episode can be reduced to the recognition of a hierarchical combination of serial and parallel episodes. For example, episode γ in Figure 4 is a serial combination of two episodes: a parallel episode δ' consisting of A and B, and an episode δ'' consisting of C alone. The occurrence of an episode in a window can be tested using such hierarchical structure: to see whether episode γ occurs in a window one checks (using a method for serial episodes) whether the subepisodes δ' and δ'' occur in this order; to check the occurrence of δ' one uses a method for parallel episodes to verify whether A and B occur.

There are, however, some complications one has to take into account. First, it is sometimes necessary to duplicate an event node to obtain a decomposition to serial and parallel episodes. Duplication works easily with injective episodes, but non-injective episodes need more complex methods. Another important aspect is that composite events have a duration, unlike the elementary events in E.

A practical alternative is to handle all episodes basically like parallel episodes, and to check the correct partial ordering only when all events are in the window. Parallel episodes can be located efficiently; after they have been found, checking the correct partial ordering is relatively fast.

4 An alternative approach: minimal occurrences

4.1 Outline of the approach

In this section we describe an alternative approach to the discovery of episodes. Instead of looking at the windows and only considering whether an episode occurs in a window or not, we now look at the exact occurrences of episodes and the relationships between those occurrences. One of the advantages of this new approach is that focusing on the occurrences of episodes allows us to more easily find rules with two window widths, one for the left-hand side and one for the whole rule, such as "if A and B occur within 15 seconds, then C follows within 30 seconds".

The approach is based on minimal occurrences of episodes. Besides the new rule formulation, the use of minimal occurrences gives raise to the following new method, called MINEPI, for the recognition of episodes in the input sequence. For each frequent episode we store information about the locations of its minimal occurrences. In the recognition phase we can then compute the locations of minimal occurrences of a candidate episode α as a temporal join of the minimal occurrences of two subepisodes of α. In addition to being simple and efficient, this formulation has the advantage that the confidences and frequencies of rules with a large number of different window widths can be obtained quickly, i.e., there is no need to rerun the analysis if one only wants to modify the window widths. In the case of complicated episodes, the time needed for recognizing the occurrence of an episode can be significant; the use of stored minimal occurrences of episodes eliminates unnecessary repetition of the recognition effort.

We identify minimal occurrences with their time intervals in the following way. Given an episode α and an event sequence $\mathbf{s}$, we say that the interval $[t_s, t_e)$ is a *minimal occurrence* of α in $\mathbf{s}$, if (1) α occurs in the window $\mathbf{w} = (w, t_s, t_e)$ on $\mathbf{s}$, and if (2) α does not occur in any proper subwindow on $\mathbf{w}$, i.e., not in any window $\mathbf{w}' = (w', t_s', t_e')$ on $\mathbf{s}$ such that $t_s \leq t_s'$, $t_e' \leq t_e$, and $width(\mathbf{w}') < width(\mathbf{w})$. The *set of (intervals of) minimal occurrences* of an episode α in a given event sequence is denoted by $mo(\alpha)$: $mo(\alpha) = \{ [t_s, t_e) \mid [t_s, t_e)$ is a minimal occurrence of $\alpha\}$.

Example 7 Consider the event sequence $\mathbf{s}$ in Figure 2 and the episodes in Figure 3. The parallel episode β consisting of event types A and B has four minimal occurrences in $\mathbf{s}$: $mo(\beta) = \{[35, 38), [46, 48), [47, 58), [57, 60)\}$. The partially ordered episode γ has the following three minimal occurrences: $[35, 39), [46, 51), [57, 62)$. □

An *episode rule* is an expression $\beta\,[win_1] \Rightarrow \alpha\,[win_2]$, where β and α are episodes such that $\beta \preceq \alpha$, and win_1 and win_2 are integers. The informal interpretation of the rule is that if episode β has a minimal occurrence at interval $[t_s, t_e)$ with $t_e - t_s \leq win_1$, then episode α occurs at interval $[t_s, t_e')$ for some t_e' such that $t_e' - t_s \leq win_2$. Formally this can be expressed in the following way. Given win_1 and β, denote $mo_{win_1}(\beta) = \{[t_s, t_e) \in mo(\beta) \mid t_e - t_s \leq win_1\}$. Further, given α and an interval $[u_s, u_e)$, define $occ(\alpha, [u_s, u_e)) =$ true if and only if there exists a minimal occurrence $[u_s', u_e') \in mo(\alpha)$ such that $u_s \leq u_s'$ and $u_e' \leq u_e$. The confidence of an episode rule $\beta\,[win_1] \Rightarrow \alpha\,[win_2]$ is now

$$\frac{|\{[t_s, t_e) \in mo_{win_1}(\beta) \mid occ(\alpha, [t_s, t_s + win_2))\}|}{|mo_{win_1}(\beta)|}.$$

Example 8 Continuing the previous example, we have, e.g., the following rules and confidences. For the rule $\beta\,[3] \Rightarrow \gamma\,[4]$ we have

$|\{[35,38),[46,48),[57,60)\}|$ in the denominator and $|\{[35,38)\}|$ in the numerator, so the confidence is 1/3. For the rule $\beta\,[3] \Rightarrow \gamma\,[5]$ the confidence is 1. $\square$

Note that since β is a subepisode of α, the rule right-hand side α contains information about the relative location of each event in it, so the "new" events in the rule right-hand side can actually be required to be positioned, e.g., between events in the left-hand side. There is also a number of possible definitions for the temporal relationship between the intervals. For instance, rules that point backwards in time can be defined in a similar way. For brevity, we only consider this one case.

We defined the frequency of an episode as the fraction of windows that contain the episode. While frequency has a nice interpretation as the probability that a randomly chosen window contains the episode, the concept is not very useful with minimal occurrences: (1) there is no fixed window size, and (2) a window may contain several minimal occurrences of an episode. Instead of frequency, we use the concept of *support*, the number of minimal occurrences of an episode: the support of an episode α in a given event sequence $\mathbf{s}$ is $|mo(\alpha)|$. Similarily to the a frequency threshold, we now use a threshold for the support: given a support threshold min_sup, an episode α is frequent if $|mo(\alpha)| \geq min\_sup$.

The current episode rule discovery task can be stated as follows. Given an event sequence $\mathbf{s}$, a class $\mathcal{E}$ of episodes, and a set W of time bounds, find all frequent episode rules of the form $\beta\,[win_1] \Rightarrow \alpha\,[win_2]$, where $\beta, \alpha \in \mathcal{E}$, $\beta \preceq \alpha$, and $win_1, win_2 \in W$.

4.2 Finding minimal occurrences of episodes

In this section we describe informally the collection Minepi of algorithms that locate the minimal occurrences of frequent serial and parallel episodes.

Let us start with some observations about the basic properties of episodes. Lemma 6 still holds: the subepisodes of a frequent episode are frequent. Thus we can use the main algorithm (Algorithm 2) and the candidate generation (Algorithm 3) also for MINEPI. We have the following results about the minimal occurrences of an episode also containing minimal occurrences of its subepisodes.

Lemma 9 Assume α is an episode and $\beta \preceq \alpha$ is its subepisode. If $[t_s, t_e) \in mo(\alpha)$, then β occurs in $[t_s, t_e)$ and hence there is an interval $[u_s, u_e) \in mo(\beta)$ such that $t_s \leq u_s \leq u_e \leq t_e$.

Lemma 10 Let α be a serial episode of size k, and let $[t_s, t_e) \in mo(\alpha)$. Then there are subepisodes α_1 and α_2 of α of size $k-1$ such that $[t_s, t_e^1) \in mo(\alpha_1)$ for some $t_e^1 < t_e$ and $[t_s^2, t_e) \in mo(\alpha_2)$ for some $t_s^2 > t_s$.

Lemma 11 Let α be a parallel episode of size k, and let $[t_s, t_e) \in mo(\alpha)$. Then there are subepisodes α_1 and α_2 of α of size $k-1$ such that $[t_s^1, t_e^1) \in mo(\alpha_1)$ and $[t_s^2, t_e^2) \in mo(\alpha_2)$ for some $t_s^1, t_e^1, t_s^2, t_e^2 \in [t_s, t_e]$, and furthermore $t_s = \min\{t_s^1, t_s^2\}$ and $t_e = \max\{t_e^1, t_e^2\}$.

The minimal occurrences of a candidate episode α are located in the following way. In the first iteration of the main algorithm, $mo(\alpha)$ is computed from the input sequence for all episodes α of size 1. In the rest of the iterations, the minimal occurrences of a candidate α are located by first selecting two suitable subepisodes α_1 and α_2 of α, and then computing a temporal join between the minimal occurrences of α_1 and α_2, in the spirit of Lemmas 10 and 11.

To be more specific, for serial episodes the two subepisodes are selected so that α_1 contains all events except the last one and α_2 in turn contains all except the first one. The minimal occurrences of α are then found with the following specification:

$$mo(\alpha) = \{\ [t_s, u_e) \quad | \quad \text{there are } [t_s, t_e) \in mo(\alpha_1) \text{ and } [u_s, u_e) \in mo(\alpha_2) \text{ such that } t_s < u_s, t_e < u_e, \text{ and } [t_s, u_e) \text{ is minimal}\}.$$

For parallel episodes, the subepisodes α_1 and α_2 contain all events except one; the omitted events must be different. See Lemma 11 for the idea of how to compute the minimal occurrences of α.

The minimal occurrences of a candidate episode α can be found in a linear pass over the minimal occurrences of the selected subepisodes α_1 and α_2. The time required for one candidate is thus $\mathcal{O}(|mo(\alpha_1)| + |mo(\alpha_2)| + |mo(\alpha)|)$, which is $\mathcal{O}(n)$, where n is the length of the event sequence. To optimize the running time, α_1 and α_2 can be selected so that $|mo(\alpha_1)| + |mo(\alpha_2)|$ is minimized.

The space requirement of the algorithm can be expressed as $\sum_i \sum_{\alpha \in \mathcal{F}_i} |mo(\alpha)|$, assuming the minimal occurrences of all frequent episodes are stored, or alternatively as $\max_i(\sum_{\alpha \in \mathcal{F}_i \cup \mathcal{F}_{i+1}} |mo(\alpha)|)$, if only the current and next levels of minimal occurrences are stored. The size of $\sum_{\alpha \in \mathcal{F}_1} |mo(\alpha)|$ is bounded by n, the number of events in the input sequence, as each event in the sequence is a minimal occurrence of an episode of size 1. In the second iteration, an event in the input sequence can start at most $|\mathcal{F}_1|$ minimal occurrences of episodes of size 2. The space complexity of the second iteration is thus $\mathcal{O}(|\mathcal{F}_1|n)$.

While minimal occurrences of episodes can be located quite efficiently, the size of the data structures can be even larger than the original database, especially in the first couple of iterations. A practical solution is to use in the beginning other pattern matching methods, e.g., similar to the ones given for WINEPI in Section 3, to locate the minimal occurrences.

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

4.3 Finding confidences of rules

We now show how the information about minimal occurrences of frequent episodes can be used to obtain confidences for various types of episode rules without looking at the data again.

Recall that we defined an episode rule as an expression $\beta\,[win_1] \Rightarrow \alpha\,[win_2]$, where β and α are episodes such that $\beta \preceq \alpha$, and win_1 and win_2 are integers. To find such rules, first note that for the rule to be frequent, the episode α has to be frequent. So rules of the above form can be enumerated by looking at all frequent episodes α, and then looking at all subepisodes β of α. The evaluation of the confidence of the rule $\beta\,[win_1] \Rightarrow \alpha\,[win_2]$ can be done in one pass through the structures $mo(\beta)$ and $mo(\alpha)$, as follows. For each $[t_s, t_e) \in mo(\beta)$ with $t_e - t_s \leq win_1$, locate the minimal occurrence $[u_s, u_e)$ of α such that $t_s \leq u_s$ and $[u_s, u_e)$ is the first interval in $mo(\alpha)$ with this property. Then check whether $u_e - t_s \leq win_2$.

The time complexity of the confidence computation for a given episode and given time bounds win_1 and win_2 is $\mathcal{O}(|mo(\beta)| + |mo(\alpha)|)$. The confidences for all win_1, win_2 in the set W of time bounds can be found, using a table of size $|W|^2$, in time $\mathcal{O}(|mo(\beta)| + |mo(\alpha)| + |W|^2)$. For reasons of brevity we omit the details.

The set W of time bounds can be used to restrict the initial search of minimal occurrences of episodes. Given W, denote the maximum time bound by $win_{max} = \max(W)$. In episode rules, only occurrences of at most win_{max} time units can be used; longer episode occurrences can thus be ignored already in the search of frequent episodes. We consider the support, too, to be computed with respect to a given win_{max}.

5 Experiments

We have run a series of experiments using WINEPI and MINEPI. The general performance of the methods, the effect of the various parameters, and the scalability of the methods are considered in this section. Consideration is also given to the applicability of the methods to various types of data sets.

The experiments have been run on a PC with 166 MHz Pentium processor and 32 MB main memory, under the Linux operating system. The sequences resided in a flat text file.

5.1 Performance overview

For an experimental overview we discovered episodes and rules in a telecommunication network fault management database. The database is a sequence of 73679 alarms covering a time period of 7 weeks. There are 287 different types of alarms with very diverse frequencies and distributions. On the average there is an alarm every minute. However, the alarms tend to occur in bursts: in the extreme cases there are over 40 alarms in one second.

We start by looking at the performance of the WINEPI method described in Section 3. There are several performance characteristics that can be used to evaluate the method. The time required by the method and the number of episodes and rules found by the method, with respect to the frequency threshold or the window width, are possible performance measures. We present results for the two opposite extreme cases of the complexity: serial episodes and injective parallel episodes.

Tables 1 and 2 represent performance statistics for finding frequent episodes in the alarm database with various frequency thresholds. The number of frequent episodes decreases rapidly as the frequency threshold increases. With a given frequency threshold, the numbers of serial and injective parallel episodes may be fairly similar, e.g., a frequency threshold of 0.002 results in

Frequency threshold	Candidates	Frequent Episodes	Iterations	Total time (s)
0.001	4528	359	45	680
0.002	2222	151	44	646
0.005	800	48	10	147
0.010	463	22	7	110
0.020	338	10	4	62
0.050	288	1	2	22
0.100	287	0	1	16

Table 1: Performance characteristics for serial episodes with WINEPI; alarm database, window width 60 s.

Frequency threshold	Candidates	Frequent Episodes	Iterations	Total time (s)
0.001	2122	185	5	49
0.002	1193	93	4	48
0.005	520	32	4	34
0.010	366	17	4	34
0.020	308	9	3	19
0.050	287	1	2	15
0.100	287	0	1	14

Table 2: Performance characteristics for injective parallel episodes with WINEPI; alarm database, window width 60 s.

151 serial episodes or 93 parallel episodes. The actual episodes are, however, very different, as can be seen from the number of iterations: recall that each iteration l produces episodes of size l. For the frequency threshold of 0.002, the longest frequent serial episode consists of 43 events (all candidates of the last iteration were infrequent), while the longest frequent injective parallel episodes have 3 events. The number of iterations equals the number of candidate generation phases. The number of database passes equals the number of iterations, or is smaller by one when there were no candidates in the last iteration.

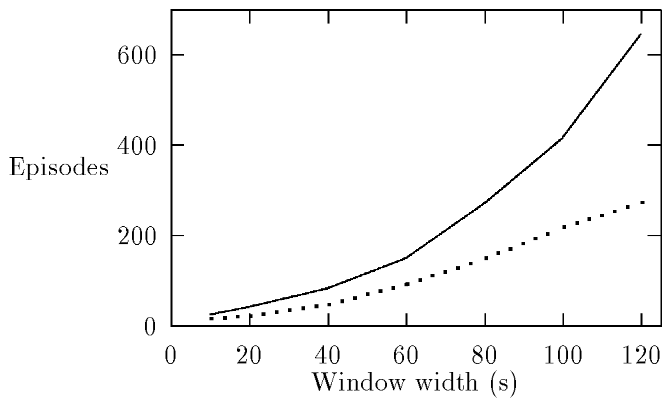

Figure 5: Number of frequent serial (solid line) and injective parallel (dotted line) episodes as a function of the window width; WINEPI, alarm database, frequency threshold 0.002.

The effect of the window width on the number of frequent episodes is represented in Figure 5. For each window width, there are considerably fewer frequent injective parallel episodes than frequent serial episodes. With the alarm data, the increase in the number of episodes is fairly even throughout the window widths that we considered. However, we will later show that this may depend heavily on the type of data we are using.

Figure 6 represents the number of serial and injective parallel episodes found by the method, and Figure 7 the total processing time required, as the frequency threshold increases. Both curves decrease steeply with the increasing frequency threshold. The time requirement is much smaller for parallel episodes than for serial episodes with the same threshold. There are two reasons for this. The parallel episodes are considerably shorter (see Tables 1 and 2) and hence, fewer database passes are needed. The complexity of recognizing injective parallel episodes is also smaller.

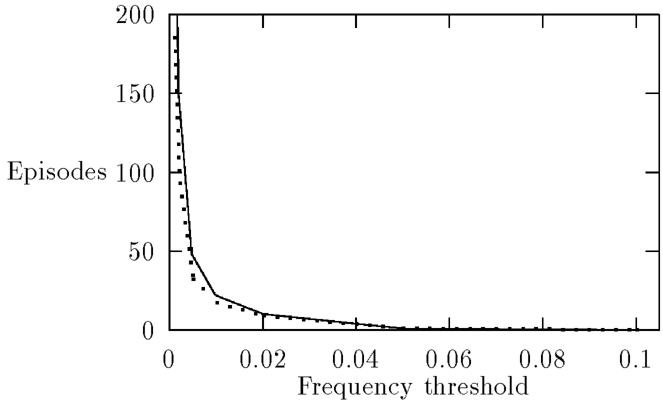

Figure 6: Number of frequent serial (solid line) and injective parallel (dotted line) episodes as a function of the frequency threshold with WINEPI; alarm database, window width 60 s.

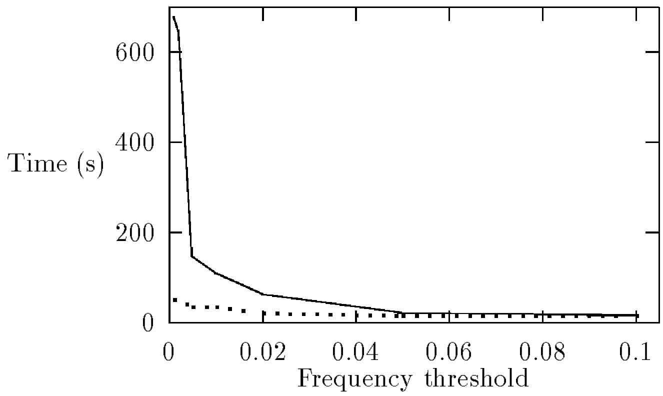

Figure 7: Processing time for serial (solid line) and injective parallel (dotted line) episodes as a function of the frequency threshold; WINEPI, alarm database, window width 60 s.

5.2 Quality of candidate generation

We now take a closer look at the candidates considered and frequent episodes found during the iterations of the procedure. As an example, let us look at what happens during the first iterations. Statistics of the first ten iterations

Episode size	Episodes	Candidates	Frequent episodes	Match
1	287	287	58	20 %
2	82369	3364	137	4 %
3	$2 \cdot 10^7$	719	46	6 %
4	$7 \cdot 10^9$	37	24	64 %
5	$2 \cdot 10^{12}$	24	17	71 %
6	$6 \cdot 10^{14}$	18	12	67 %
7	$2 \cdot 10^{17}$	13	12	92 %
8	$5 \cdot 10^{19}$	13	8	62 %
9	$1 \cdot 10^{22}$	8	3	38 %
10	$4 \cdot 10^{24}$	3	2	67 %

Table 3: Number of candidate and frequent serial episodes during the first ten iteration phases with WINEPI; alarm database, frequency threshold 0.001, window width 60 s.

of a run with a frequency threshold of 0.001 and a window width of 60 s is shown in Table 3.

The three first iterations dominate the behavior of the method. During these phases, the number of candidates is large, and only a small fraction (less than 20 per cent) of the candidates turns out to be frequent. After the third phase the candidate generation is efficient, few of the candidates are found infrequent, and although the total number of iteration phases is 45, the last 35 iterations involve only 1–3 candidates each. Thus we could safely combine several of the later iteration steps, to reduce the number of database passes.

If we take a closer look at the frequent episodes, we observe that all frequent episodes longer than 7 events consist of repeating occurrences of two very frequent alarms. Each of these two alarms occurs in the database more than 12000 times (16 per cent of the events each).

Support threshold	Candidates	Frequent Episodes	Iterations	Total time (s)
50	12732	2735	83	28
100	5893	826	71	16
250	2140	298	54	16
500	813	138	49	14
1000	589	92	48	14
2000	405	64	47	13
4000	352	53	46	12

Table 4: Performance characteristics for serial episodes with MINEPI; alarm database, maximum time bound 60 s.

Support threshold	Candidates	Frequent Episodes	Iterations	Total time (s)
50	10041	4856	89	30
100	4376	1755	71	20
250	1599	484	54	14
500	633	138	49	13
1000	480	89	48	12
2000	378	66	47	12
4000	346	53	46	12

Table 5: Performance characteristics for parallel episodes with MINEPI; alarm database, maximum time bound 60 s.

5.3 Comparison of algorithms WINEPI and MINEPI

Tables 4 and 5 represent performance statistics for finding frequent episodes with MINEPI, the method using minimal occurences. Compared to the corresponding figures for WINEPI in Tables 1 and 2, we observe the same general tendency for a rapidly decreasing number of candidates and episodes, as the support threshold increases.

The episodes found by WINEPI and MINEPI are not necessarily the same. If we compare the cases in Tables 1 and 4 with approximately the same number of frequent episodes, e.g., 151 serial episodes for WINEPI and 138 for

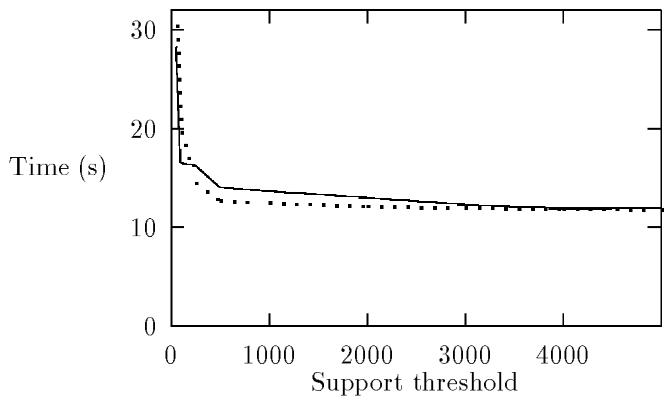

Figure 8: Processing time for serial (solid line) and injective parallel (dotted line) episodes with MINEPI; alarm database, maximum time bound 60 s.

MINEPI, we notice that they do not correspond to the same episodes. The sizes of the longest freuquent episodes are somewhat different (43 for the original, 48 for the minimal occurrence method). The frequency threshold 0.002 for WINEPI corresponds, at the minimum, to about 150 instances of the episode, while the support threshold used for MINEPI is 500. The difference between the methods is very clear for small episodes. Consider an episode α consisting of just one event A. WINEPI considers a single event A to occur in 60 windows of width 60 s, while MINEPI sees only one minimal occurrence. On the other hand, two successive events of type A result in α occuring in 61 windows, but the number of minimal occurrences is doubled from 1 to 2.

Figure 8 shows the time requirement for finding frequent episodes with MINEPI. The processing time for MINEPI reaches a plateau when the size of the maximal episodes no longer changes (in this case, at support threshold 500). The behavior is similar for serial and parallel episodes. The time requirements of MINEPI should not be directly compared to WINEPI: the episodes discovered are different, and our implementation of MINEPI works entirely in the main memory. With very large databases this might not be

Varying support threshold, four time bounds

Support threshold	Distinct rules	Rule gen. time (s)
50	50470	149
100	10809	29
250	4041	20
500	1697	16
1000	1221	15
2000	1082	14
4000	1005	14

Varying number of time bounds, support threshold 1000

Number of time bounds	All rules	Rule gen. time (s)
1	1221	13
2	2488	13
4	5250	15
10	11808	18
20	28136	22
30	42228	27
60	79055	43

Table 6: Number of rules and rule generation time with MINEPI; alarm database, serial episodes, support threshold 1000, maximum time bound 60 s, confidence threshold 0.

possible during the first iterations; either the minimal occurrences need to be stored on the disk, or other methods (e.g., variants of Algorithms 4 and 5) must be used.

5.4 Rules

The methods can easily produce very large amounts of rules. Recall that rules are constructed by considering all frequent episodes α as the right-hand side and all subepisodes $\beta \preceq \alpha$ as the left-hand side of the rule. Additionally, MINEPI considers variations of these rules with all the time bounds in the given set W.

Table 6 represents results with serial episodes. The initial episode generation with MINEPI took around 14 s, and the total number of frequent episodes was 92. The table shows the number of rules obtained by MINEPI with confidence threshold 0 and with maximum time bound 60 s. On the left, with a varying support threshold, rules that differ only in their time bounds are excluded from the figures; the rule generation time is, however, obtained by generating rules with four different time bounds.

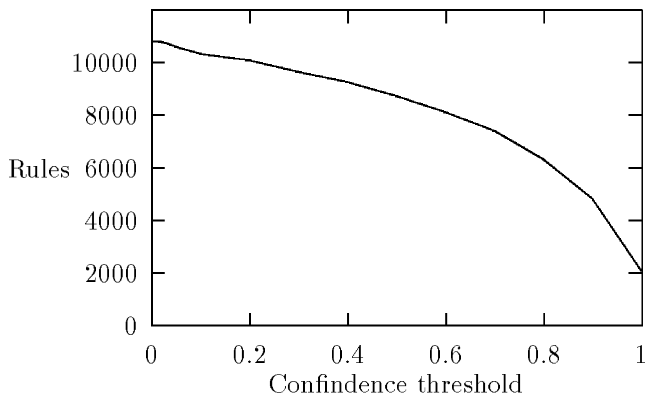

Figure 9: Total number of distinct rules found by MINEPI with various confidence thresholds; alarm database, maximum time bound 60 s, support threshold 100.

The minimal occurrence method is particularly useful, if we are interested in finding rules with several different time bounds. The right side of Table 6 represents performance results with a varying number of time bounds. The time requirement increases slowly as more time bounds are used, and the time increases slowlier than the number of rules.

The amount of almost 80000 rules, obtained with 60 time bounds, may seem unnecessarily large and unjustified. Remember, however, that there are only 1221 distinct rules. The rest of the rules present different combinations of time bounds, in this case down to the granularity of one second. For the cost of 43 s we thus obtain very fine-grained rules from our frequent episodes. Different criteria can then be used to select the most interesting rules from these. Figure 9 represents the effect of the confidence threshold to the number of distinct rules found by MINEPI. Although the initial number of rules may be quite large, it decreases fairly rapidly if we require a reasonable confidence.

Data set name	Events	Event types	Supp. thr.	Max time b.	Conf. thr.	Freq. epis.	Rules
alarms	73679	287	100	60	0.8	826	6303
WWW	116308	7634	250	120	0.2	454	316
text1	5417	1102	20	20	0.2	127	19
text2	2871	905	20	20	0.2	34	4
protein	4941	22	7	10		21234	

Table 7: Characteristic parameter values for each of the data sets and the number of episodes and rules found by MINEPI.

5.5 Results with different data sets

In addition to the experiments on the alarm database, we have run MINEPI on a variety of different data collections to get a better view of the usefulness of the method. The data collections that were used, some typical parameter values for them, and some results are presented in Table 7.

The WWW data is part of the WWW server log from the Department of Computer Science at the University of Helsinki. The log contains requests to WWW pages at the department's server; such requests can be made by WWW browsers at any host in the Internet. We consider the WWW page fetched as the event type. The number of events in our data set is 116308, covering three weeks in February and March, 1996. In total, 7634 different pages are referred to. Requests for images have been excluded from consideration.

Suitable support thresholds vary a lot, depending on the number of events and the distribution of event types. A suitable maximum time bound for the device generated alarm data is one minute, while the slower pace of a human user requires using a larger time bound (two minutes or more) for the WWW log. By using a relatively small time bound we reduce the probability of unrelated requests contributing to the support. A low confidence threshold for the WWW log is justified since we are interested in all fairly usual patterns of usage, not only in the dominating ones. In the WWW server log we found, e.g., long often used paths of pages from the home page of the department to the pages of individual courses. Such behavior suggests that rather than using a bookmark directly to the home page of a course, many users quickly navigate there from the departmental home page.

The two text data collections are modifications of the same English text. Each word is considered an event, and the words are indexed consecutively to give a "time" for each event. The end of each sentence causes a gap in the indexing scheme, to correspond to a longer distance between words in different sentences. We used text from GNU man pages (the gnu awk manual). The size of the original text (text1) is 5417 words, and the size of the condensed text file (text2), where noninformative words such as articles, prepositions, and conjunctions, have been stripped off, is 2871 words. The number of different words in the original and the condensed text is 1102, resp. 905.

For text analysis, there is no point in using large time bounds, since it is unlikely that there is any connection between words that are not fairly close to each other. This can be clearly seen in Figure 10 which represents the number of episodes found on various window widths using WINEPI. This figure reveals behavior that is distinctively different from the corresponding Figure 5 for the alarm database. We observe that for the text data, the window widths from 24 to 50 produce practically the same amount of serial episodes. The number of episodes will only increase with considerably larger window widths. For this data, the interesting frequent episodes are smaller than 24, while the episodes found with much larger window widths are noise. The same phenomenon can be observed for parallel episodes.

Only few rules can be found in text using a simple analysis like this. The strongest rules in the original text involve either the word "`gawk`", or

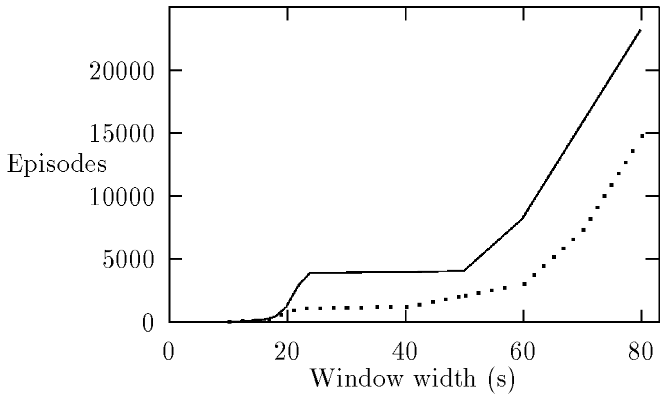

Figure 10: Number of serial (solid line) and injective parallel (dotted line) episodes as a function of the window width; WINEPI, compressed text data (text2), frequency threshold 0.02.

common phrases such as

$$\texttt{the}, \texttt{value}\,[2] \;\Rightarrow\; \texttt{of}\,[3] \qquad \text{(confidence 0.90)}$$

meaning that in 90 % of the cases where the words "`the value`" are consecutive, they are immediately followed by the preposition "`of`". These rules were not found in the condensed text since all prepositions and articles have been stripped off. The few rules in the condensed text contain multiple occurrences of the word "`gawk`", or combinations of words occurring in the header of each man page, such as "`free software`".

We performed scale-up tests with 5, 10, and 20 fold multiples of the compressed text file, i.e., sequences of approximately 2900 to 58000 events. The results in Figure 11 show that the time requirement is roughly linear with respect to the length of the input sequence, as could be expected.

Finally, we experimented with protein sequences. We used data in the PROSITE database [1] of the ExPASy WWW molecular biology server of the Geneva University Hospital and the University of Geneva [11]. PROSITE contains biologically significant DNA and protein patterns that help to

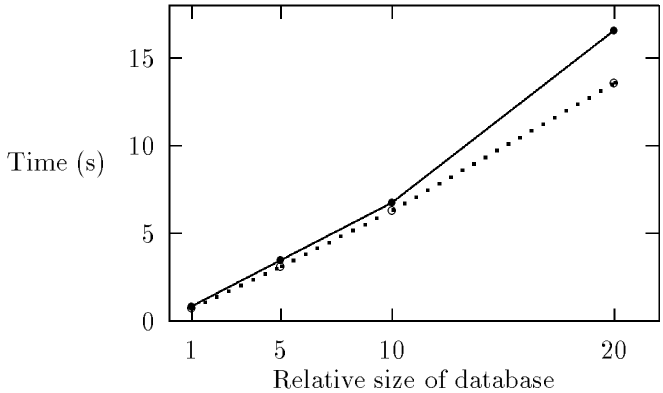

Figure 11: Scale-up results for serial (solid line) and injective parallel (dotted line) episodes with MINEPI; compressed text data, maximum time bound 60, support threshold 10 for the smallest file (n-fold for the larger files).

identify to which family of protein (if any) a new sequence belongs. The purpose of our experiment is to evaluate our algorithm against an external data collection and patterns that are known to exist, not to find patterns previously unknown to the biologists. We selected as our target a family of 7 sequences ("DNA mismatch repair proteins 1", PROSITE entry PS00058). The sequences in the family are known to contain the string `GFRGEAL` of seven consequtive symbols. We transformed the data in a manner similar to the English text: symbols are indexed consecutively, and between the protein sequences we place a gap. The total length of this data set is 4941 events, with an alphabet of 22 event types. The method could be easily modified to take several separage sequences as input, and to compute the support of an episode α, e.g., as the number of input sequences that contain a (minimal) occurrence of α of length at most the maximum time bound.

The parameter values for the protein database are chosen on purpose to reveal the pattern that is known to be present in the database. The window width was selected to be 10, i.e., slightly larger than the length of the pattern that we were looking for, and the support threshold was set to 7, for the seven individual sequences in the original data. With this data, we are only interested in the longest episodes (of length 7 or longer). Of the more than 20000 episodes found, 17 episodes are of length 7 or 8. As expected, these contain the sequence `GFRGEAL` that was known to be in the database. The longer episodes are variants of this pattern with an eighth symbol fairly near, but not necessarily immediately subsequent to the pattern (e.g., `GFRGEAL*S`). These types of patterns belong to the pattern class used in PROSITE but, to our suprise, these longer patterns are not reported in the PROSITE database.

6 Extensions and related work

The task of discovering frequent parallel episodes can be stated as a task of discovering all frequent sets, a central phase of discovering assocation rules [2], the rule generation methods are also basically the same for association rules and WINEPI. The levelwise main algorithm has also been used successfully in the search of frequent sets [3].

Technical problems related to the recognition of episodes have been researched in several fields. Taking advantage of the slowly changing contents of the group of recent events has been studied, e.g., in artificial intelligence, where a similar problem in spirit is the many pattern/many object pattern match problem in production system interpreters [9]. Also, comparable strategies using a sliding window have been used, e.g., to study the locality of reference in virtual memory [7]. Our setting differs from these in that our window is a queue with the special property that we know in advance when an event will leave the window; this knowledge is used by WINEPI in the recognition of serial episodes. In MINEPI, we take advantage of the fact that we know where subepisodes of candidates have occurred.

The recent work on sequence data in databases (see [21]) provides interesting openings towards the use of database techniques in the processing of queries on sequences. A problem similar to the computation of frequencies occurs also in the area of active databases. There triggers can be specified as composite events, somewhat similar to episodes. In [10] it is shown how finite automata can be constructed from composite events to recognize when a trigger should be fired. This method is not practical for episodes since the deterministic automata could be very large.

The methods for matching sets of episodes against a sequence have some similarities to the algorithms used in string matching (e.g., [12]). In particular, recognizing serial episodes in a sequence can be seen as locating all occurrences of subsequences, or matches of patterns with variable length don't care symbols, where the length of the occurrences is limited by the window width. Learning from a set of sequences has received considerable interest in the field of bioinformatics, where an interesting problem is the discovery of patterns common to a set of related protein or amino acid sequences. The classes of patterns differ from ours; they can be, e.g., substrings with fixed length don't care symbols [15]. Closer to our patterns are those considered in [24]. The described algorithm finds patterns that are similar to serial episodes; however, the patterns have a given minimum length, and the occurrences can be within a given edit distance. Recent results on the pattern matching aspects of recognizing episodes can be found in [6].

The work most closely related to ours is perhaps [4]. There multiple sequences are searched for patterns that are similar to the serial episodes with some extra restrictions and an event taxonomy. Our methods can be extended with a taxonomy by a direct application of the similar extensions to association rules [13, 14, 22]. Also, our methods can be applied on analyzing several sequencies; there is actually a variety of choices for the definition of frequency of an episode in a set of sequencies. More recently, the pattern class of [4] has been extended with windowing, some extra time constraints, and an event taxonomy [23]. — For a survey on patterns in sequential data, see [17].

In stochastics, event sequence data is often called a marked point process [16]. It should be noted that traditional methods for analyzing marked point processes are ill suited for the cases where the number of event types is large. However, there exists an interesting combination of techniques: frequent episodes are discovered first, and then the phenomena they describe are analyzed in more detail with methods for marked point processes.

There are also some interesting similarities between the discovery of frequent episodes and the work done on inductive logic programming (see, e.g., [20]); a noticeable difference is caused by the sequentiality of the underlying data model, and the emphasis on time-limited occurrences. Similarly, the problem of looking for one occurrence of an episode can be viewed as a constraint satisfaction problem.

The class of patterns discovered can be easily modified in several directions. Different windowing strategies could be used, e.g., considering only windows starting every win' time units for some win', or windows starting from every event. Other types of patterns could also be searched for, e.g., substrings with fixed length don't care symbols; searching for episodes in several sequences is no problem. A more general framework for episode discovery has been presented in [18]. There episodes are defined as combinations of events satisfying certain user specified unary of binary conditions.

7 Conclusions

We presented a framework for discovering frequent episodes in sequential data. The framework consists of defining episodes as partially ordered sets of events, and looking at windows on the sequence. We described an algorithm, WINEPI, for finding all episodes from a given class of episodes that are frequent enough. The algorithm was based on the discovery of episodes by only considering an episode when all its subepisodes are frequent, and on incremental checking of whether an episode occurs in a window. The implementation shows that the method is efficient. We have applied the method in the analysis of the alarm flow from telecommunication networks, and discovered episodes have been embedded in alarm handling software.

We also presented an alternative approach, MINEPI, to the discovery of frequent episodes, based on minimal occurrences of episodes. This approach supplies more power for representing connections between events, as it produces rules with two time bounds.

Both rule formalisms have their advantages. While the rules of MINEPI are often more informative, the frequencies and confidences of the rules of WINEPI have nice interpretations as probabilities concerning randomly chosen windows. For a large part the algorithms are similar, there are significant differences only in the computation of the frequency or support. Roughly, a general tendency in the performance is that WINEPI can be more efficient in the first phases of the discovery, mostly due to smaller space requirement. In the later iterations, MINEPI is likely to outperform WINEPI clearly. The methods can be modified for cross-use, i.e., WINEPI for finding minimal occurrences and MINEPI for counting windows, and for some large problems — whether the rule type of WINEPI or MINEPI — a mixture of the two methods could give better performance than either alone.

Interesting extensions to the work presented here are facilities for rule querying and compilation, i.e., methods by which the user could specify the episode class in high-level language and the definition would automatically be compiled into a specialization of the algorithm that would take advantage of the restrictions on the episode class. Other open problems include the combination of episode techniques with marked point processes and intensity models.

References

[1] B. A., B. P., and H. K. The PROSITE database, its status in 1995. *Nucleic Acids Research*, 24:189 – 196, 1995.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD'93)*, pages 207 – 216, May 1993.

[3] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307 – 328. AAAI Press, Menlo Park, CA, 1996.

[4] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering (ICDE'95)*, pages 3 – 14, Taipei, Taiwan, Mar. 1995.

[5] C. Bettini, X. S. Wang, and S. Jajodia. Testing complex temporal relationships involving multiple granularities and its application to data mining. In *Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'96)*, pages 68 – 78, Montreal, Canada, June 1996.

[6] G. Das, R. Fleischer, L. Gasieniec, D. Gunopulos, and J. Kärkkäinen. Episode matching. Manuscript, 1997.

[7] P. J. Denning. The working set model of program behavior. *Communications of the ACM*, 11(5):323 – 333, 1968.

[8] C. Dousson, P. Gaborit, and M. Ghallab. Situation recognition: Representation and algorithms. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 166 – 172, Chambery, France, Aug. 1993.

[9] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17 – 37, 1982.

[10] N. Gehani, H. Jagadish, and O. Shmueli. Composite event specification in active databases. In *Proceedings of the Eightteenth International Conference on Very Large Data Bases (VLDB'92)*, pages 327 – 338, Vancouver, Canada, Aug. 1992.

[11] Geneva University Hospital and University of Geneva, Switzerland. *ExPASy Molecular Biology Server.* http://expasy.hcuge.ch/.

[12] R. Grossi and F. Luccio. Simple and efficient string matching with k mismatches. *Information Processing Letters*, 33:113 – 120, 1989.

[13] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB'95)*, pages 420 – 431, Zurich, Swizerland, 1995.

[14] M. Holsheimer, M. Kersten, H. Mannila, and H. Toivonen. A perspective on databases and data mining. In *Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD'95)*, pages 150 – 155, Montreal, Canada, Aug. 1995.

[15] I. Jonassen, J. F. Collins, and D. G. Higgins. Finding flexible patterns in unaligned protein sequences. *Protein Science*, 4(8):1587 – 1595, 1995.

[16] J. D. Kalbfleisch and R. L. Prentice. *The Statistical Analysis of Failure Time Data.* John Wiley Inc., New York, NY, 1980.

[17] P. Laird. Identifying and using patterns in sequential data. In K. Jantke, S. Kobayashi, E. Tomita, and T. Yokomori, editors, *Algorithmic Learning Theory, 4th International Workshop*, pages 1 – 18, Berlin, 1993. Springer-Verlag.

[18] H. Mannila and H. Toivonen. Discovering generalized episodes using minimal occurrences. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96)*, pages 146 – 151, Portland, Oregon, Aug. 1996. AAAI Press.

[19] R. A. Morris, W. D. Shoaff, and L. Khatib. An algebraic formulation of temporal knowledge for reasoning about recurring events. In *Proceedings of the International Workshop on Temporal Representation and Reasoning (TIME-94)*, pages 29 – 34, Pensacola, FL, May 1994.

[20] S. Muggleton. *Inductive Logic Programming*. Academic Press, London, 1992.

[21] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: Design & implementation of a sequence database system. In *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB'96)*, pages 99 – 110, Mumbay, India, Sept. 1996.

[22] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB'95)*, pages 407 – 419, Zurich, Swizerland, 1995.

[23] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Advances in Database Technology—5th International Conference on Extending Database Technology (EDBT'96)*, pages 3 – 17, Avignon, France, 1996.

[24] J. T.-L. Wang, G.-W. Chirn, T. G. Marr, B. Shapiro, D. Shasha, and K. Zhang. Combinatorial pattern discovery for scientific data: Some preliminary results. In *Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD'94)*, pages 115 – 125, June 1994.