Proceedings of The First International Conference on Knowledge Discovery and Data Mining (KDD-95), pages 210 – 215, Montreal, Canada, August 20–21, 1995.

Discovering frequent episodes in sequences
Extended abstract

Heikki Mannila and **Hannu Toivonen** and **A. Inkeri Verkamo**
Department of Computer Science
P.O.Box 26, FIN–00014 University of Helsinki, Finland
{mannila, htoivone, verkamo}@cs.Helsinki.FI

Abstract

Sequences of events describing the behavior and actions of users or systems can be collected in several domains. In this paper we consider the problem of recognizing frequent episodes in such sequences of events. An episode is defined to be a collection of events that occur within time intervals of a given size in a given partial order. Once such episodes are known, one can produce rules for describing or predicting the behavior of the sequence. We describe an efficient algorithm for the discovery of all frequent episodes from a given class of episodes, and present experimental results.

Introduction

Most machine learning and data mining techniques are adapted towards the analysis of unordered collections of data. However, there are important application areas where the data to be analyzed has an inherent sequential structure.

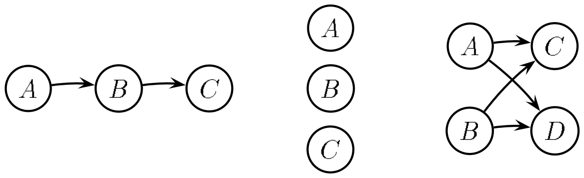
For instance, in telecommunications network monitoring or empirical user interface studies it is easy to log a lot of information about the behavior and actions of the user and the system. Abstractly such a log can be viewed as a sequence of events, where each event has an associated time of occurrence. An example of an event sequence is

$$(A, 123), (B, 125), (D, 140), (A, 150), (C, 151),$$
$$(B, 155), (D, 201), (A, 220), (D, 222), (B, 225).$$

Here A, B, and C are event types (e.g., different types of notifications in the network or different types of user actions), and the numbers indicate the times when the events occurred.

One basic problem in analyzing such a sequence is to find frequent *episodes*, i.e., collections of events occurring frequently close to each other. For example, in the above sequence the episode "A followed by B" occurs several times, even when the sequence is viewed through a window of (say) 6 time units.

Episodes, in general, are partially ordered sets of events. They can also be described as directed acyclic graphs in the obvious way. Consider, for instance, the episodes (a), (b), and (c):



(a) (b) (c)

Episode (a) is a *serial episode*: it occurs in a sequence only if there are events A, B, and C that occur in this order in the sequence relatively close together. Note that in the sequence there can be other events occurring between these three. Episode (b) is a *parallel episode*: no constraints on the relative order of A, B, and C are given. Episode (c) occurs in a sequence if there are occurrences of A and B and these precede the occurrences of C and D; no constraints on the relative order of A and B (or C and D) are given. All four occurrences must be close to each other in time.

The user defines how close is close enough by giving the width of the *time window* within which the episode must occur. The user also specifies how often an episode has to occur to be considered frequent.

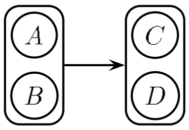
In the analysis of sequences the user is interested in finding all frequent episodes from a class of episodes. Such a class can be, e.g., all serial episodes, all parallel episodes, or all episodes where the partial order matches some part of the network topology; one can even consider the class of all partial orders.

Once the episodes are known, they can be used to obtain rules for prediction. For example, if we know that the serial episode (a) occurs in 4.0 % of the windows and that the serial subepisode consisting only of A and B occurs in 4.2 % of the windows, we can estimate that after seeing a window where A and B occur in this order, there is a chance of 0.95 that C will occur in the same window. One can compute such strengths of rules using the information obtained by our algorithm; it is also easy to obtain confidence intervals for the rule strengths.

Discovery of frequent episodes In this paper we consider the following problem. Given a class of episodes, an input sequence of events, a window width, and a frequency threshold, find all episodes of the class that occur frequently enough.

We describe a general algorithm for solving this task. The algorithm has two alternating phases: building new candidate episodes, and evaluating how often these occur in the sequence. The efficiency of the algorithm is based on three observations:

1. While there are potentially a very large number of episodes that have to be checked, the search space can be pruned by building larger episodes from smaller ones in a certain way. If the serial episode ABC is frequent, with the same window width also the serial subepisodes AB, BC, and AC are frequent. This holds in general: all subepisodes are at least as frequent as the superepisode. Our algorithm utilizes this observation in the opposite direction: it is only necessary to test the occurrences of episodes whose all subepisodes are frequent.
2. Episodes can be recognized efficiently by "sliding" a window on the input sequence. Typically two adjacent windows have a lot of overlap, and are therefore very similar to each other. We take advantage of this similarity: after recognizing episodes in a window, we make incremental updates in our data structures to recognize the episodes that occur in the next window.
3. Recognition of a complex episode can be reduced to the recognition of simple ones: every episode can be seen as a recursive combination of parallel and serial episodes. For example, the episode (c) above is a serial combination of two episodes: (d.1), a parallel episode consisting of A and B, and (d.2), a parallel episode consisting of C and D:



(d.1) (d.2)

The occurrence of an episode in a window can be tested by using this structure and specialized methods for the recognition of parallel and serial episodes. To see whether episode (c) occurs, we check (using the method for serial episodes) whether (d.1) and (d.2) occur in this order; to check the occurrence of (d.1) we use the method for parallel episodes to verify whether A and B occur.

We have implemented our method and used it in the analysis of telecommunication network alarm data. The discovered episodes have been considered useful by domain experts.

Related work For an excellent survey on patterns in sequential data, see (Laird 1993). Wang *et al.* consider the problem of discovering patterns from a number of input sequences. Their algorithm finds patterns that are similar to serial episodes, with respect to given minimum length, edit distance, and frequency threshold. They also consider probabilistic pruning of patterns by using sampling (Wang *et al.* 1994).

The idea of building candidate patterns from smaller ones has been profitably used in the discovery of association rules (Mannila, Toivonen, & Verkamo 1994; Agrawal & Srikant 1994; Agrawal *et al.* 1995), and occurs also in other contexts (e.g. (Klösgen 1995; Wang *et al.* 1994)).

Taking advantage of the slowly changing contents of the group of recent events in the recognition phase has been studied under different fields. Within AI, a similar problem in spirit is the many pattern/many object pattern match problem in production system interpreters (see e.g. (Forgy 1982)). In active databases a related problem is the efficient detection of trigger events (see e.g. (Gehani, Jagadish, & Shmueli 1992)). Also, comparable strategies using a sliding window have been used e.g. to study locality of reference in virtual memory (Denning 1968).

The methods for matching sets of episodes against a sequence have some similarities to the algorithms used in string matching (see (Aho 1990) for a survey). Similarly, the work on learning regular languages from strings has some similarities to our work. However, in string matching or learning regular languages one typically is interested in finding substrings where there may not be interleaved events within an occurrence of an episode.

Formulation

Event sequences Given a class E_0 of elementary *event types*, an *event* is a pair (e, t), where $e \in E_0$ and t is an integer. An *event sequence* $\mathcal{S}$ is a triple $(T_\mathcal{S}, T^\mathcal{S}, S)$, where $T_\mathcal{S}$ is the starting time, $T^\mathcal{S}$ is the closing time, and S is an ordered sequence of events, i.e.,

$$S = (e_1, t_1), (e_2, t_2), \ldots, (e_n, t_n),$$

where $e_i \in E_0$ and $T_\mathcal{S} \leq t_i < T^\mathcal{S}$ for all $i = 1, \ldots, n$, and $t_i \leq t_{i+1}$ for all $i = 1, \ldots, n-1$.

A *window* on event sequence $\mathcal{S} = (T_\mathcal{S}, T^\mathcal{S}, S)$ is an event sequence $\mathcal{W} = (T_\mathcal{W}, T^\mathcal{W}, W)$, where $T_\mathcal{S} \leq T_\mathcal{W}, T^\mathcal{W} \leq T^\mathcal{S}$, and W consists of those pairs (e_i, t_i) from S where $T_\mathcal{W} \leq t_i < T^\mathcal{W}$. The width of the window $\mathcal{W}$ is $width(\mathcal{W}) = T^\mathcal{W} - T_\mathcal{W}$. Given an event sequence $\mathcal{S}$ and an integer w, we denote by $aw(\mathcal{S}, w)$ the set of all windows $\mathcal{W}$ on $\mathcal{S}$ such that $width(\mathcal{W}) = w$. There are $T^\mathcal{S} - T_\mathcal{S} - w + 1$ such windows. (Also other windowing strategies can be used, e.g., only windows starting every w' time units for some w' could be considered, or windows starting from every event.)

Episodes An (elementary) *episode* $\varphi = (V, \leq, g)$ is a set of nodes V, a partial order $\leq$ on V, and a mapping $g : V \to E_0$ 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 $\leq$.

We generalize this definition to allow for recursive decomposition and recognition of episodes. A *composite* episode is an episode $\varphi = (V, \leq, g)$ such that the mapping g is a mapping $V \to E$, where the set of event types $E = E_0 \cup \Sigma$, and Σ is the set of all composite episodes. For example, the decomposition (d) of episode (c) above is a composite episode consisting of two episodes; both these are composite (actually also elementary) episodes consisting of two nodes in parallel.

The episode φ is *parallel* if the partial order relation $\leq$ is trivial (i.e., $x \not\leq y$ for all $x \neq y$). The episode φ is *serial* if the partial order relation $\leq$ is a total order.

The crucial observation is that any elementary episode can be equivalently described as a composite episode built recursively from parallel and serial (composite) episodes. Writing $s(\varphi_1, \ldots, \varphi_n)$ and $p(\varphi_1, \ldots, \varphi_n)$ for the serial and parallel episodes consisting of episodes $\varphi_1, \ldots, \varphi_n$, we can write the episode (c) above as $s(p(A, B), p(C, D))$.[1]

An episode $\varphi' = (V', \leq', g')$ is a *subepisode* of $\varphi = (V, \leq, g)$, $\varphi' < \varphi$, if $V' \subset V$, for all $v \in V'$, $g'(v) = g(v)$, and for all $v, w \in V'$ with $v \leq' w$ also $v \leq w$. An episode φ' is an *immediate subepisode* of φ, $\varphi' <_0 \varphi$, if $\varphi' < \varphi$ and there is no episode φ'' such that $\varphi' < \varphi'' < \varphi$. An episode φ is a *superepisode* of φ' if $\varphi' < \varphi$, and *immediate superepisode* if $\varphi' <_0 \varphi$. A class of episodes $\mathcal{C}$ is *closed*, if for each episode φ in $\mathcal{C}$ all subepisodes $\varphi' < \varphi$ are in $\mathcal{C}$ as well.

Frequent episodes An elementary episode $\varphi = (V, \leq, g)$ *occurs* in an event sequence

$$\mathcal{S} = (T_\mathcal{S}, T^\mathcal{S}, ((e_1, t_1), (e_2, t_2), \ldots, (e_n, t_n))),$$

if there exists an injective mapping $h : V \to \{1, \ldots, n\}$ such that $g(v) = e_{h(v)}$ for all $v \in V$, and for all $v, w \in V$ with $v \leq w$, $h(v) < h(w)$. This definition can be generalized for composite episodes; we need the generalization only for serial and parallel composite episodes. A parallel episode $\varphi = (V, \leq, g)$ occurs in an event sequence $\mathcal{S}$ if all episodes in $g(V)$ occur disjointly in $\mathcal{S}$. A serial episode $\varphi = (V, \leq, g)$ occurs in an event sequence $\mathcal{S}$ if there exist mappings $h_1, h_2 : V \to \{1, \ldots, n\}$ such that for all $v \in V$ the subepisode $g(v)$ occurs in the event sequence $(t_{h_1(v)}, t_{h_2(v)} + 1, ((e_{h_1(v)}, t_{h_1(v)}), \ldots, (e_{h_2(v)}, t_{h_2(v)})))$ and for all $v, w \in V$ with $v \leq w$ we have $h_2(v) < h_1(w)$.

[1] It is sometimes necessary to duplicate an event node to obtain this decomposition; note the similarity to series-parallel graphs. Consider, for instance, the partial order $A \leq C, B \leq C, B \leq D$, where either node B or C must be duplicated.

$C_1 := \{\{e\} \mid e \in E_0\}$;
$i := 1$;
while $C_i \neq \emptyset$ **do**
  recognition: read the sequence $\mathcal{S}$, and let L_i be the frequent elements of C_i with respect to *min_fr*;
  building: compute $C_{i+1} \subset \mathcal{C}$ from L_i;
  $i := i + 1$;
od;
for all i, output L_i;

Figure 1: Main algorithm.

If an episode φ occurs in $\mathcal{S}$, we write $\mathcal{S} \models \varphi$. The *frequency* of φ in the set $aw(\mathcal{S}, w)$ of all windows on $\mathcal{S}$ of size w is

$$fr(\varphi, \mathcal{S}, w) = \frac{|\{\mathcal{W} \in aw(\mathcal{S}, w) \mid \mathcal{W} \models \varphi\}|}{|aw(\mathcal{S}, w)|}.$$

Given a threshold *min_fr* for the frequency, a given episode φ is *frequent* if $fr(\varphi, \mathcal{S}, w) \geq$ *min_fr*. If φ is frequent then all its subepisodes are frequent.

The problem is now the following. Given a sequence $\mathcal{S}$, a closed class $\mathcal{C}$ of episodes, a window width w, and a frequency threshold *min_fr*, find all frequent episodes from $\mathcal{S}$.

Algorithm

In Figure 1 we give an algorithm for finding all frequent episodes $\varphi \in \mathcal{C}$ in the given event sequence $\mathcal{S}$, given a closed class $\mathcal{C}$ of episodes and a frequency threshold *min_fr*.

The algorithm works iteratively, alternating between building and recognition phases. First, in the building phase of an iteration i, a collection C_i of new *candidate episodes* of i elementary events is built, using the information available from smaller frequent episodes. Then, these candidate episodes are recognized in the event sequence and their frequencies are computed. The collection L_i consists of frequent episodes in C_i. In the next iteration $i+1$, candidate episodes in C_{i+1} are built using the information about the frequent episodes in L_i. The algorithm starts by constructing C_1 to contain all episodes consisting of single elementary events. In the end, the frequent episodes in L_i for all i are output.

Building new episodes

Remember that all subepisodes φ' of any frequent episode φ are also frequent. That is actually the specification of candidate episodes: in each iteration, candidates are all those episodes of size i whose all subepisodes are frequent.

A method to compute the candidate collection C_i is to first generate episodes φ of size i, and then to test that all immediate subepisodes $\varphi' <_0 \varphi$ are in L_{i-1} (see (Mannila, Toivonen, & Verkamo 1994; Agrawal &

Srikant 1994; Agrawal *et al.* 1995) for similar ideas). The generation phase can be implemented efficiently e.g. by taking pairs of episodes from L_{i-1} that overlap in $i-2$ nodes.

Recognizing episodes in sequences

Episodes are recognized in sequences in an incremental fashion. Two adjacent windows $\mathcal{W}_i = (T_i, T_i + w, W_i)$ and $\mathcal{W}_{i+1} = (T_i + 1, T_i + w + 1, W_{i+1})$ are typically very similar to each other. We take advantage of this: after recognizing episodes in $\mathcal{W}_i$, we make incremental updates in our data structures to achieve the *shift* of the window to obtain $\mathcal{W}_{i+1}$.

Remember that any elementary episode can be viewed as a composite episode consisting of only (composite) parallel and serial episodes. In the recognition of episodes in sequences, we recursively decompose elementary episodes to parallel and serial composite episodes. To check whether a parallel episode $\varphi = p(\varphi_1, \ldots, \varphi_n)$ occurs, we apply the method below for parallel episodes; to check whether a serial episode $\varphi = s(\varphi_1, \ldots, \varphi_n)$ occurs, we apply the method below for serial episodes.[2]

Parallel episodes For simplicity, we will consider parallel episodes $\varphi = (V, \leq, g)$ where the mapping g is an injection, i.e. there are no two events of the same type in an episode.

For each such candidate parallel episode φ we maintain a counter $\varphi.count$ that indicates how many events of φ are present in the window. When $\varphi.count$ becomes equal to $|\varphi|$ (indicating that φ is entirely included in the window) we save the index of the window in $\varphi.inwindow$. When $\varphi.count$ decreases again (indicating that φ is no longer entirely in the window) we increase the field $\varphi.occur$ by the number of windows where φ remained entirely in the window.

To carry this out efficiently, the collection of candidate episodes is organized so that for each event $e \in E$ the episodes containing e are linked together to a list *contains(e)*. When the window is shifted and the set of events in the window changes, the corresponding lists are traversed and the counters of the episodes updated.

The requirement that the mapping g of a parallel episode $\varphi = (V, \leq, g)$ has to be an injection can be easily removed, allowing multisets to be presented.

Serial episodes Serial candidate episodes are recognized by using state automata that accept the candidate episodes and ignore all other input. We initialize a new instance of the automaton for a serial episode φ every time the first event of φ comes into the window; the automaton is removed when the same event leaves the window. When an automaton for φ reaches its accepting state (indicating that φ is entirely included in the window), and if there are no other automata for φ in the accepting state already, we save the index of the window in $\varphi.inwindow$. When an automaton in accepting state is removed, and if there are no other automata for φ in the accepting state (indicating that φ is no longer entirely in the window) we increase the field $\varphi.occur$ by the number of windows where φ remained entirely in the window.

Note that it is useless to have multiple automata that are in the same state. It suffices to maintain the one that reached the state last since it will be also removed last. There are thus at most $|\varphi|$ automata for an episode φ. For each automata we need to know the window index at which the automata was initialized. We represent all the automata for φ with two arrays of size $|\varphi|$: one represents φ itself and contains its events, the other array contains the window indices of the automata.

To access and traverse the automata efficiently they are organized in the following way. For each event type $e \in E$, the automata that accept e are linked together to a list *waits(e)*. When an event (e, t) enters the window during a shift, the list *waits(e)* is traversed and a transition is performed in each automaton. If an automaton reaches a common state with another automaton, the older one is removed (i.e., overwritten). For each event (e, t) in the window, the automata initialized at t are linked to a list *beginsat(t)*. When the event is removed from the window all automata on the list *beginsat(t)* are removed.[3]

Analysis of time complexity For simplicity, suppose that the candidate episodes φ are all of the same size, that the class of event types E is fixed, and assume that exactly one event takes place every time unit. Denote by m the length of the event sequence; there and thus $m - w + 1$ windows, i.e., $\mathcal{O}(m)$.

For parallel episodes, consider w successive shifts of one time unit. During such sequence of shifts, each of the k candidate episodes φ can undergo at most $2|\varphi|$ changes: any event of an episode φ can be dropped out or taken into the window at most once. This is due to the fact that any event taken in cannot be dropped out during the next w time units. Supposing the class of event types E is fixed, finding the change in the set of events during a shift of the window takes constant time.

[2] An interesting alternative approach to the recognition, in which decomposition is not necessary, is to use inverse structures. For each frequent episode $\varphi' \in L_i$ we save the indexes presenting the windows $\mathcal{W}$ for which $\mathcal{W} \models \varphi'$. Then, in the recognition phase, for a candidate episode $\varphi \in C_{i+1}$ we can compute $\{\mathcal{W} \mid \mathcal{W} \models \varphi\} = \bigcap\{\mathcal{W}' \mid \mathcal{W}' \models \varphi' \text{ and } \varphi' <_0 \varphi\}$. This holds for all but serial episodes, for which some additional information is needed (Mannila & Toivonen 1995; Toivonen 1995).

[3] An obvious practical improvement to the approach above is to combine similar prefixes of episodes and avoid maintaining redundant information. An alternative approach altogether would be to handle serial episodes basically like parallel episodes, and to check the correct ordering only when all events are in the window.

w (s)	Serial episodes		Parallel episodes	
	$\Sigma\|L_i\|$	time (s)	$\Sigma\|L_i\|$	time (s)
10	16	31	10	8
20	31	63	17	9
40	57	117	33	14
60	87	186	56	15
80	145	271	95	21
100	245	372	139	21
120	359	478	189	22

Table 1: Characteristics of runs with a fixed frequency threshold $min\_fr = 0.003$ and a varying window width w.

Reading the input takes time m. The total time complexity is thus $\mathcal{O}(\frac{m}{w}k|\varphi| + m)$. For a trivial nonincremental method where the sequence is preprocessed into windows, the time requirement for recognizing k candidate episodes φ in m windows, plus the time required to read in m windows of size w, is $\mathcal{O}(mk|\varphi| + mw)$, i.e., larger by a factor of w.

For serial episodes, the input sequence consists in the worst case of events of only one event type, and there is a serial episode φ of events of that same type. Potentially every shift of the window results now in an update in every prefix. There are m shifts, k episodes, and $|\varphi|$ prefixes for each episode; the time to read the input is m. The worst case time complexity is thus $\mathcal{O}(mk|\varphi| + m)$. This is close to the complexity of the trivial nonincremental method $\mathcal{O}(mk|\varphi| + mw)$. In practical situations, however, the time requirement is considerably smaller, and we approach the savings obtained in the parallel case.

Experimental results

We have applied our methods to a telecommunications network fault management database. The database consists of 73679 alarms covering a time period of 50 days. We examined the sequence of alarm types; there were 287 different types with diverse frequencies and distributions.

The experiments have been run on a PC with 90 MHz Pentium processor and 16 MB main memory, under the Linux operating system. The sequence of (alarm type, time) pairs resided in a flat text file.

Performance overview Tables 1 and 2 give an overview of the discovery of frequent episodes in an empirical setting. In Table 1, serial and parallel episodes have been discovered with a fixed frequency threshold $min\_fr = 0.003$ and a varying window width w; in Table 2, episodes have been discovered with a fixed window width $w = 60$ s and a varying frequency threshold min_fr. $\Sigma|L_i|$ denotes the total number of frequent episodes.

These experiments show that our approach is efficient. Running times are between 5 seconds and 8 minutes, in which time hundreds of frequent episodes could be found. Relatively small changes in the parameters influence the number of episodes found, but the method is robust in the sense that a change in one parameter does not result in the replacement of any of the episodes found.

	Serial episodes		Parallel episodes	
min_fr	$\Sigma\|L_i\|$	time (s)	$\Sigma\|L_i\|$	time (s)
0.1	0	7	0	5
0.05	1	12	1	5
0.008	30	62	19	14
0.004	60	100	40	15
0.002	150	407	93	22
0.001	357	490	185	22

Table 2: Characteristics of runs with a fixed window width $w = 60$ s and a varying frequency threshold min_fr.

size i	HS	$\|C_i\|$	$\|L_i\|$	Match
1	287	287.0	30.1	11 %
2	82369	1078.7	44.6	4 %
3	$2 \cdot 10^7$	192.4	20.0	10 %
4	$7 \cdot 10^9$	17.4	10.1	58 %
5	$2 \cdot 10^{12}$	7.1	5.3	74 %
6	$6 \cdot 10^{14}$	4.7	2.9	61 %
7	$2 \cdot 10^{17}$	2.9	2.1	75 %
8	$5 \cdot 10^{19}$	2.1	1.7	80 %
9	$1 \cdot 10^{22}$	1.7	1.4	83 %
10–		17.4	16.0	92 %

Table 3: Number of serial candidate and frequent episodes per iteration with frequency threshold $min\_fr = 0.003$, averaged over window widths $w = 10, 20, 40, 60, 80, 100$, and 120 s. HS is the size of the hypothesis space; 'match' is the fraction $|L_i|/|C_i|$.

Quality of candidate generation Table 3 presents the number of serial candidate and frequent episodes per iteration averaged over a number of test runs.

In the first iteration, for size $i = 1$, all 287 elementary events have to be checked. The larger the episodes become, the more combinatorial information there exists to take advantage of. From size 4 up, over one half of the candidates turned out to be frequent.[4]

Incremental recognition Figure 2 presents the ratio of times needed for trivial vs. incremental recognition of candidate episodes. The figure shows that

[4] Another possible improvement is to combine iterations by generating candidate episodes for several iterations at once, and thus avoid reading the input sequence so many times. This pays off in the later iterations, where there are otherwise only few candidates to recognize, and where the match is good.

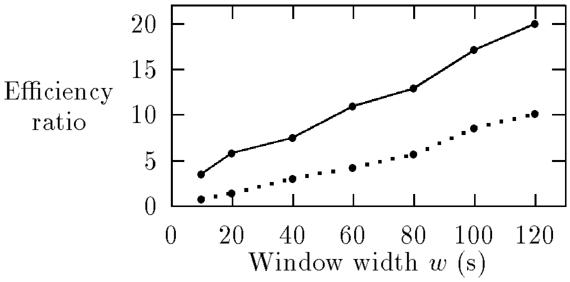


Figure 2: Ratio of times needed for trivial vs. incremental recognition methods for parallel (solid line) and serial episodes (dotted line) as functions of window width w.

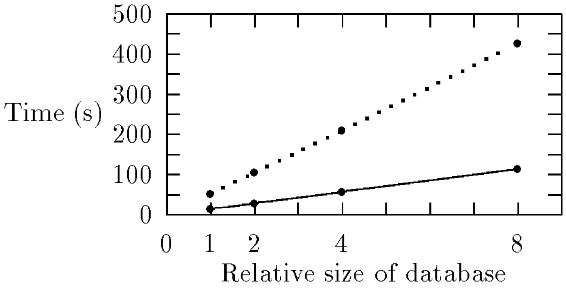


Figure 3: Scale-up results for parallel (solid line) and serial episodes (dotted line) with $w = 60$ s and $min\_fr = 0.01$.

the incremental methods are faster by a factor of 1 – 20, approximately linearly with respect to the window width (10 – 120). This matches the worst case analysis of recognizing parallel episodes, and indicates that the incremental recognition method is useful in practice also for serial episodes.

Scale-up We performed scale-up tests with 1 to 8 fold multiples of the database, i.e. sequences with 74 to 590 thousand events. Figure 3 shows that the time requirement is linear with respect to the length of the input sequence, as could be expected from the analysis.

Conclusions

We have presented a framework for discovering frequent episodes in sequential data. The framework consists of defining episodes as partial orders, and looking at windows of the sequence. We described an algorithm 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 a telecommunications network. Discovered episodes have been considered useful by domain experts.

We are currently investigating the use of inverse structures in the recognition phase of the algorithm. The preliminary results are quite encouraging.

References

Agrawal, R., and Srikant, R. 1994. Fast algorithms for mining association rules in large databases. In *20th International Conference on Very Large Databases (VLDB 94).*

Agrawal, R.; Mannila, H.; Srikant, R.; Toivonen, H.; and Verkamo, A. I. 1995. Fast discovery of association rules. In Fayyad, U. M.; Piatetsky-Shapiro, G.; Smyth, P.; and Uthurusamy, R., eds., *Advances in Knowledge Discovery and Data Mining.* AAAI Press. To appear.

Aho, A. V. 1990. Algorithms for finding patterns in strings. In van Leeuwen, J., ed., *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity.* Elsevier. 255–400.

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

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

Gehani, N.; Jagadish, H.; and Shmueli, O. 1992. Event specification in an active object-oriented database. In *1992 International Conference on Management of Data (SIGMOD 92)*, 81 – 90.

Klösgen, W. 1995. Efficient discovery of interesting statements in databases. *Journal of Intelligent Information Systems* 4(1):53 – 69.

Laird, P. 1993. Identifying and using patterns in sequential data. In *Algorithmic Learning Theory, 4th International Workshop*, 1 – 18. Springer-Verlag.

Mannila, H., and Toivonen, H. 1995. Discovering frequent episodes in sequences using inverse structures. Manuscript in preparation.

Mannila, H.; Toivonen, H.; and Verkamo, A. I. 1994. Efficient algorithms for discovering association rules. In *AAAI Workshop on Knowledge Discovery in Databases (KDD 94)* , 181 – 192.

Toivonen, H. 1995. Ph.D. thesis in preparation.

Wang, J. T.-L.; Chirn, G.-W.; Marr, T. G.; Shapiro, B.; Shasha, D.; and Zhang, K. 1994. Combinatorial pattern discovery for scientific data: Some preliminary results. In *1994 International Conference on Management of Data (SIGMOD 94)*, 115 – 125.