International Journal of Uncertainty,
Fuzziness and Knowledge-Based Systems
Vol. 16, No. 5 (2008) 645–661
© World Scientific Publishing Company

MINING FREQUENT TEMPORAL PATTERNS IN INTERVAL SEQUENCES

STEFFEN KEMPE[*], JOCHEN HIPP[†] and CARSTEN LANQUILLON[‡]
Daimler AG, Group Research, 89081 Ulm, Germany
[*] *Steffen.Kempe@daimler.com*
[†] *Jochen.Hipp@daimler.com*
[‡] *Carsten.Lanquillon@daimler.com*

RUDOLF KRUSE
Dept. of Knowledge Processing and Language Engineering,
Otto-von-Guericke-University of Magdeburg,
39106 Magdeburg, Germany
Kruse@iws.cs.uni-magdeburg.de

Received 20 February 2007
Revised 28 January 2008

Recently a new type of data source came into the focus of knowledge discovery from temporal data: interval sequences. In contrast to event sequences, interval sequences contain labeled events with a temporal extension. However, existing algorithms for mining patterns from interval sequences proved to be far from satisfying our needs. In brief, we missed an approach that, at the same time, defines support as the number of pattern instances, allows input data that consists of more than one sequence, implements time constraints on a pattern instance, and counts multiple instances of a pattern within one interval sequence. In this paper we propose a new support definition which incorporates these properties. We also describe FSMSet, an algorithm that employs the new support definition, and demonstrate its performance on field data from the automotive business.

Keywords: Temporal data mining; frequent patterns; interval sequences.

1. Introduction

Mining sequences from temporal data is a well known data mining task which gained much attention in the past, e.g. Refs. 1–5. In all these approaches, the temporal data is considered to consist of events. Each event has a label and a timestamp. In the following, we want to focus on temporal data where an event has a temporal extension. These temporally extended events are called temporal intervals. Each temporal interval can be described by a triplet (b, e, l) where b and e denote the beginning and the end of the interval and l its label. For example a sequence of temporal intervals may describe the medical history of a patient in a hospital or

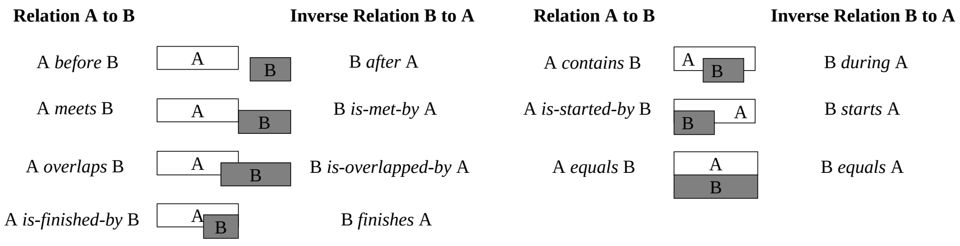


Fig. 1. Allen's interval relations.

the data collected by a flight recorder. Early work by Kam and Fu,[6] Hppner,[7] Papapetrou *et al.*[8] and Winarko[9] employed Allen's interval relations (Fig. 1) to find frequent patterns in sequences of temporal intervals.

At Daimler we are interested in analyzing sequences of temporal intervals in order to further extend the knowledge about our products, to improve customer satisfaction, or to assist the development process. In the following, we will describe three application examples from different business areas which can all be represented by using temporal intervals.

Application one: Quality monitoring of a vehicle fleet. A major task for any vehicle manufacturer is to monitor the quality of the product in the field. Temporal data mining can support this task by identifying sequences of faults or combinations of faults and vehicle configurations which occur more often than others. In this case one interval sequence describes the history of one vehicle. The configuration of a vehicle, e.g. whether it is an estate car or a limousine, can be described by temporal intervals. The build date is the beginning and the current day is the end of such a temporal interval. Other temporal intervals contain information about garage stopovers or the installation of additional equipment. The frequent patterns from a set of interval sequences (i.e. from a vehicle fleet) can be used by an engineer to introduce product changes if necessary.

Application two: Analytical customer relationship management. Many of our customers are re-buyers, i.e. they buy a vehicle, keep it for a certain period of time, sell it, and buy a new vehicle. These customers are especially valuable as they show a high brand loyalty. Mining the information that we have about these "good" customers might help us to determine where we have cross- and upselling potential for other customers. The information about the customers comes from two sources. The first source is the sale itself. Here the vehicle type (model line, special option codes, etc.) and sales type (whether leasing or full buy was preferred) are of particular interest. The second source are questionnaires which were sent to the customers. The questionnaires range from micro-economic questions (age, size of household, income, etc.) to customer satisfaction (repair shops, marketing, etc.). The patterns found are useful to guide marketing or commercial actions.

Application three: Mining CAN-Bus data. The popularity of electronic systems (navigation, mobile phones, antiblock system, etc.) has led to a steady increase in the number of electronic control units (ECU) within a vehicle. Most of the ECUs are communicating with each other over a CAN-Bus System (Controller Area Network). During the development of new ECUs or a new model line the network traffic of the CAN-Bus is particularly analyzed as it contains the status and all status changes of the connected ECUs. There is also information about the general driving conditions available (e.g. speed, gear, steering angle, etc.). All these information can be expressed by using interval sequences. Then one interval sequence belongs to one trip of one vehicle. Mining these sequences for frequent patterns helps to identify typical driving situations for the ECUs. In case of an ECU malfunction temporal patterns can also support electronic diagnostics by pointing out the conditions and their relations under which the malfunction occurred.

Despite their different application areas, all three examples share a common problem setting. There are instances (vehicles, customers, trips) which are described by a sequence of temporal intervals. The mining task is to find all frequent temporal patterns within these interval sequences.

2. Related Work and Structure

Previous investigations on discovering patterns from interval sequences include the work of Höppner,[7] Kam and Fu,[6] Papapetrou *et al.*,[8] and Winarko and Roddick.[9] These approaches can be divided into two different groups.

The main difference between both groups is the definition of support. Höppner defines the *temporal support of a pattern*. This definition is closely related to the *frequency* in Ref. 3. The temporal support can be interpreted as the probability to see an instance of the pattern within a time window if the time window is randomly placed on the interval sequence.

All other approaches count the number of instances for each pattern. The pattern counter is incremented once for each sequence that contains the pattern. If an interval sequence contains multiple instances of a pattern then these additional instances will not further increment the counter. This way of counting instances of a pattern was introduced in Ref. 1 and is also employed in Refs. 2, 12–14.

The rest of this paper is organized as follows. In the next section we provide formal definitions of the mining task. In Sec. 4 we argue that the commonly used definitions for the support of temporal patterns are not feasible for our application examples. Therefore we introduce a new support definition which is motivated by our experience in the automotive industry, but is directly transferable to other domains. Next, we propose FSMSet a novel algorithm for finding frequent patterns which implements the new support definition. The algorithm is evaluated on real world data from our domain in Sec. 6.

3. Foundations

As mentioned above we represent a temporal interval as a triplet (b, e, l). Next, temporal intervals are used to define interval sequences.

Definition 1. (Temporal Interval) Given a set of labels L, we say the triplet $(b, e, l) \in \mathbb{R} \times \mathbb{R} \times L$ is a temporal interval, if $b \leq e$. The set of all temporal intervals over L is denoted by I.

Definition 2. (Interval Sequence) Given a sequence of temporal intervals, we say $(b_1, e_1, l_1), (b_2, e_2, l_2), \ldots, (b_n, e_n, l_n)$ with $(b_i, e_i, l_i) \in I$ is an interval sequence, if

$$\forall (b_i, e_i, l_i), (b_j, e_j, l_j) \in I, i \neq j : b_i \leq b_j \wedge e_i \geq b_j \Rightarrow l_i \neq l_j \tag{1}$$

and

$$\begin{gathered}\forall (b_i, e_i, l_i), (b_j, e_j, l_j) \in I, i < j : \\ (b_i < b_j) \vee (b_i = b_j \wedge e_i < e_j) \vee (b_i = b_j \wedge e_i = e_j \wedge l_i < l_j)\end{gathered} \tag{2}$$

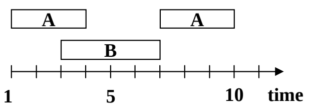
hold. A given set of interval sequences is denoted by $\mathbb{S}$.

Equation (1) above is referred to as the *maximality assumption*.[7] The maximality assumption guarantees that each temporal interval A is maximal, in the sense that there is no other temporal interval in the sequence sharing a time with A and carrying the same label. Equation (2) requires that an interval sequence has to be ordered by the begin (primary), end (secondary) and label (tertiary, lexicographically) of its temporal intervals.

Without temporal extension there are only two possible relations. One event is before (or after as the inverse relation) the other or they coincide. Due to the temporal extension of temporal intervals the possible relations between two intervals become more complex. There are 7 possible relations (respectively 13 including inverse relations). These interval relations have been described by Allen in Ref. 15 and are depicted in Fig. 1. Each relation of Fig. 1 is a temporal pattern on its own that consists of two temporal intervals. Patterns with more than two temporal intervals are straightforward. One just needs to know which interval relation exists between each pair of labels. Using the set of Allen's interval relations $\mathbb{I}$, a temporal pattern is defined by:

Definition 3. (Temporal Pattern) A pair $P = (s, R)$, where $s : (1, \ldots, n) \to L$ and $R \in \mathbb{I}^{n \times n}$, $n \in \mathbb{N}$, is called a "temporal pattern of size n".

Figure 2(a) shows an example of an interval sequence. The corresponding temporal pattern is given in Fig. 2(b). The temporal pattern in Fig. 2(b) is represented by using a table. While the table directly corresponds to $R \in \mathbb{I}^{n \times n}$ in Definition 3 the function $s : (1, \ldots, n) \to L$ is given by the order of the column headers.

(a)


(b)

	A	B	A
A	e	o	b
B	io	e	m
A	a	im	e

Fig. 2. (a) Example of an Interval Sequence: (1,4,A), (3,7,B), (7,10,A). (b) Example of a Temporal Pattern (e stands for *equals*, o for *overlaps*, b for *before*, m for *meets*, io for *is-overlapped-by*, etc.)

Note that a temporal pattern needs not necessarily be valid in the sense that it must be possible to construct an interval sequence for which the pattern holds true. On other hand, if a temporal pattern holds true for an interval sequence we consider this sequence as an instance of the pattern.

Definition 4. (Instance) A temporal pattern $P = (s, R)$ holds true for an interval sequence $S = (b_i, e_i, l_i)_{1 \leq i \leq n}$, if $\forall i, j : s(i) = l_i \wedge s(j) = l_j \wedge R[i,j] = \mathrm{ir}([b_i, e_i], [b_j, e_j])$ with function ir returning the relation between two given intervals. We say that the interval sequence S is an instance of temporal pattern P. We say that an interval sequence S' *contains* an instance of P if $S \subseteq S'$, i.e. S is a subsequence of S'.

Obviously a temporal pattern can only be valid if its labels have the same order as their corresponding temporal intervals have in an instance of the pattern.

The mining task is to find all temporal patterns in a given set of interval sequences which satisfy a user specified minimum support threshold. Note that this mining task is closely related to frequent itemset mining.[16–19]

4. A New Support Definition

After having conducted extensive experiments with the algorithms of Höppner and Papapetrou *et al.* we came to the conclusion that contemporary algorithms do not meet our needs. The problem originates from a gap between the support definitions of temporal patterns in Refs. 6, 8, 9 or 7 and the demands of our application.

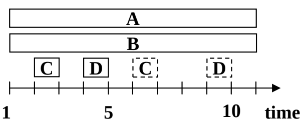
On one hand, Höppner's algorithm handles only a single interval sequence using *temporal support* as support definition. Despite the fact that we had multiple sequences, we found it hard to interpret the support of the frequent patterns. The temporal support gives the probability of seeing an instance of the pattern in a randomly placed time window. In general, this probability is not related to the number of instances of the pattern. It could either be a few instances that are visible for a long time or a lot of instances that are only shortly visible in the time window that lead to the same temporal support. Yet the accurate number of pattern instances is indispensable for generating knowledge in our domain.

On the other hand, the algorithms of Papapetrou *et al.*,[8] Kam and Fu,[6] and Winarko and Roddick[9] only count one instance of a pattern within the same interval sequence. Multiple instances in an interval sequence are neglected. In our applications typically several instances of a pattern can occur within one interval sequence.

In addition we missed time constraints on a pattern instance as one instance should not be arbitrarily long.

Consider the pattern *C before D* in the example of Fig. 3(a). As the interval sequence contains an instance of the pattern, the support of Papapetrou *et al.*, Kam and Fu, and Winarko and Roddick is 1. Using a time window of size 3, Höppner will calculate an absolute support of 3 (a duration of 2 for the first occurrence and 1 for

(a)



(b)

	A	B	C	D
A	e	***e***	***c***	***c***
B	e	e	***c***	***c***
C	d	d	e	***b***
D	d	d	a	e

Fig. 3. (a) Example of an Interval Sequence. (b) Example of a Temporal Pattern (e is the abbreviation for *equals*, c for *contains*, etc.)

the second).[a] For our applications we would like to see an support of 2 as *C before D* occurs twice in the data. In general, we think that counting pattern occurrences is the only feasible support definition when mining the repair history of vehicles or CAN-Bus data. A domain expert has to review all resulting patterns before any further actions can take place. Obviously the most important information for the domain expert is how often a certain pattern occurred in the data.

Summing up, for our needs and comparable applications neither of the previous approaches turned out to be satisfying. Thus we developed a new support definition which:

(1) counts the number of pattern instances,
(2) handles multiple instances of a pattern within one interval sequence,
(3) and allows time constraints on a pattern instance.

In Ref. 3, Mannila *et al.* introduced *minimal occurrences* as a support definition for patterns in a single sequence of events. We extend the approach of minimal occurrences to meet the demands of temporal intervals and Allen's interval relations.

Definition 5. (Minimal Occurrence) For a given interval sequence S a time interval (time window) $[b, e]$ is called a minimal occurrence of the k-Pattern P ($k \geq 2$), if

(1) the time interval $[b, e]$ of S contains an instance of P, and
(2) there is no proper subinterval $[b', e']$ of $[b, e]$ which also contains an instance of P.

For a given interval sequence S a time interval $[b, e]$ is called a minimal occurrence of the 1-Pattern P, if

(1) the temporal interval (b, e, l) is contained in S, and
(2) l is the Label in P.

The definition above contains a special case for temporal patterns of size 1, i.e. P contains only one label. Then every temporal interval (b, e, l) with $b < e$ leads to

[a]For Höppner's temporal support the absolute support must be divided by the sum of the sequence length (10) and the window width (3). Hence the temporal support in the example is $\frac{3}{10+3}$.

an infinite number of minimal occurrences.[b] Therefore the minimal occurrences of P are $[b, e]$ for each temporal interval (b, e, l) and l is the label of P.

Minimal occurrences also provide an easy way to introduce time constraints on a pattern instance. Suppose a pattern instance is only valid if it occurs within a certain period of time, then we just need to count those minimal occurrences whose lengths do not exceed the time limit.

After we introduced minimal occurrences in our application, we realized that the support definition is still not sufficient. There is a subset of temporal patterns whose supports should be calculated in a different way. Figure 3(a) gives an example of such a temporal pattern (solid lines). The minimal occurrence is $[1, 11]$. In any smaller time window, the relation *equals* between the temporal intervals A and B is not visible. Thus if an interval sequence contains a second C *before* D during the temporal intervals A and B (dashed lines in Fig. 3(a)), it will not be counted as it produces the same minimal occurrence.

This example is important to us because in the quality monitoring application the temporal intervals C and D might describe garage stopovers for a car with A and B specifying its vehicle configuration. So we want to count multiple instances of temporal patterns (here C *before* D) for different combinations of configurations. Hence, in the example above, we have to count the minimal occurrences of the subpattern[c] C *before* D given that the subpattern is contained in temporal intervals A and B.

As a result, we have to identify all temporal patterns where a subpattern is decisive for the support calculation. In these patterns exists a subpattern which is contained in all other temporal intervals of the temporal pattern. We can decide whether a given temporal pattern P contains such a subpattern by transforming the problem into graph theory based on the upper triangular matrix of its relation table. We start by creating an empty graph. For each label of the temporal pattern we insert a vertex into the graph. Next we create an edge for each relation in the upper triangular matrix of the relation table which is not *contains* between the corresponding vertices in the graph. If the resulting graph is unconnected then there is a subpattern which is contained in all other temporal intervals of the pattern. This subpattern corresponds to one of the connected subgraphs.

We call a temporal pattern connected if its graph is *connected*. Otherwise we call the temporal pattern *unconnected*. In contrast to Ref. 3 we define the support of a connected pattern P as its total number of minimal occurrences in all sequences of $\mathbb{S}$. If P is unconnected the support is given by the total number of minimal occurrences of its subpattern.

[b]In this case each time window $[x, x]$ with $x \in R$ and $b \leq x \leq e$ would be a minimal occurrence.
[c]A subpattern can easily be obtained by removing one or more labels and their corresponding rows and columns from the superpattern's relation table.

5. New Algorithm: FSMSet

A pattern is considered to be frequent if its support is above a user defined support threshold ($\geq$ *MinSupp*). As in existing approaches, the main idea is to generate all frequent temporal patterns by applying the Apriori scheme of candidate generation and support evaluation. This approach requires that each subpattern of a frequent temporal pattern be frequent. Unfortunately the extension of minimal occurrences to temporal intervals destroys this downward closure property. The problem arises if a temporal interval is used as the same part of a pattern for multiple instances. Consider, e.g., the interval sequence (1,11,A), (2,3,C), (7,8,C) (see Fig. 3(a)). The two minimal occurrences of the pattern A *contains* C are $[2, 3]$ and $[7, 8]$ but there is only one minimal occurrence of A $[1, 11]$. Here A is used twice as the same part of the pattern. Hence, the downward closure property is not guaranteed for minimal occurrences.

A closer investigation shows that *contains* is the only relation for which the property does not hold. Consider the pattern A *overlaps* B. The downward closure property can only fail if A overlaps two or more Bs. This is impossible as these Bs would have to share a time interval which is prohibited by the maximality assumption (Equation (1)). The same argumentation holds for *meets*, *is-finished-by*, *is-started-by*, *equals* and their inverse relations. In case of A *before* B there can be several Bs after the A but the definition of minimal occurrences allows only to count the first B.

Obviously the downward closure property only fails for unconnected temporal patterns. Our approach to solve this problem is by treating connected and unconnected temporal patterns differently.

5.1. *Candidate generation*

Following the scheme of Apriori,[17] our algorithm consists of two main steps: generation of candidate sets and support evaluation of these candidates. These two steps are alternately repeated until no more candidates are generated. Apriori starts with the frequent 1-patterns and then successively derives all k-candidates from the set of all frequent (k-1)-patterns.

In general this generation and test scheme exploits the downward closure property of support in two ways:

(a) Generating the set of all (k+1)-candidates by joining all pairs of frequent k-patterns that share a common (k-1)-pattern as their first part ensures that we obtain a superset of the frequent (k+1)-patterns.
(b) Those candidate patterns can be pruned a priori to evaluation of support values for which at least one subpattern is known to be not frequent.

Yet, when mining frequent patterns in the context of temporal intervals, the situation is not as straightforward. As mentioned above, the downward closure property does no longer hold for unconnected patterns. For example, when

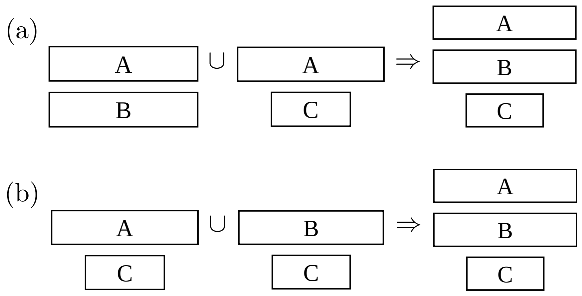


Fig. 4. Two different ways of generating the temporal Pattern A *equals* B, A and B *contain* C: (a) by the common subpattern A or (b) by the common subpattern C.

generating candidates in the Apriori-way, the candidate pattern A *equals* B, A *and* B *contain* C would be generated based on the two subpatterns A *equals* B and A *contains* C (see Fig. 4(a)). However A *equals* B needs not necessarily be frequent even when the resulting candidate A *equals* B, A *and* B *contain* C is frequent. As a result we face two challenges: Not only is pruning the candidates by infrequent subsets no longer possible but also does the Apriori-approach for candidate generation no longer guarantee that we do not miss any of the frequent patterns.

In the example above, obviously the subpatterns A *contains* C and B *contains* C are necessarily frequent when the candidate itself is frequent (see Fig. 4(b)). Generally, in a valid unconnected (k+1)-pattern, $k \geq 2$, there exists a $j, 1 \leq j \leq k$, such that the first j labels always describe those temporal intervals which contain all other temporal intervals of the pattern (labels $j+1, \ldots, k+1$). *Contains* implies "starts before and ends after", so the ordering of sequences guarantees exactly this property. Hence, in the opposite way, the last $k+1-j$ labels of the pattern are always responsible for the frequency of the pattern. If we remove the first or second label from a frequent (k+1)-pattern the resulting k-patterns must still be frequent. In other words, generating (k+1)-candidates by joining the two subpatterns that share the last k–1 labels ensures that the set of generated (k+1)-candidates is a superset of the frequent (k+1)-patterns. We only have to guarantee that the initial set of frequent 2-patterns contains all frequent unconnected patterns.

Thus, by modifying the candidate generation step, we can transfer the completeness property of the Apriori approach, see (a) above, to unconnected temporal patterns. The interaction between candidate generation and support evaluation is shown in Algorithm 1. In detail, our approach of candidate generation is as follows: The candidate patterns of size 1 are generated by using all available labels in the dataset. In the next step, we use all 1-candidates to create the candidate patterns of size 2. This guarantees that we will find all frequent 2-patterns, albeit they are connected or unconnected. For the actual candidate generation and test approach the frequent patterns of size k ($k \geq 2$) are used to generate the candidate patterns of size k+1. This is achieved by joining every pair of temporal patterns P and Q which are identical w.r.t. the last k-1 rows and columns of their relation table, i.e.

Algorithm 1 FSMSet: procedure to find all frequent temporal patterns

```
procedure FindFrequentPatterns($\mathbb{S}$, $L$)        ▷ $\mathbb{S}$: interval sequences, $L$: set of labels in $\mathbb{S}$
    k := 1, $C_k := \{l \in L\}$                          ▷ $C_i$: candidate patterns in $i$-th iteration
    repeat
        evaluateSupport($C_k$, $\mathbb{S}$)
        for all $c \in C_k$ do
            if getSupport($c$) $\geq$ MinSupp then
                $F_k := F_k \cup \{c\}$                    ▷ $F_i$: frequent patterns in $i$-th iteration
        end for
        if k = 1 then $C_2$ := generateCandidates($C_1$)
        else $C_{k+1}$ := generateCandidates($F_k$)
        k := k+1
    until $C_k = \emptyset$
    return $F_{1\ldots k}$
end procedure
```

$$
\begin{array}{c||c|c}
P & l_1^p & l_2 \ldots l_k \\
\hline\hline
l_1^p & \text{e} & \text{C} \\
\hline
l_2 \, \vdots \, l_k & \text{B} & \text{A}
\end{array}
\;\cup\;
\begin{array}{c||c|c}
Q & l_1^q & l_2 \ldots l_k \\
\hline\hline
l_1^q & \text{e} & \text{E} \\
\hline
l_2 \, \vdots \, l_k & \text{D} & \text{A}
\end{array}
\;\Longrightarrow\;
\begin{array}{c||c|c|c}
R & l_1^p & l_1^q & l_2 \ldots l_k \\
\hline\hline
l_1^p & \text{e} & ? & \text{C} \\
\hline
l_1^q & \neg? & \text{e} & \text{E} \\
\hline
l_2 \, \vdots \, l_k & \text{B} & \text{D} & \text{A}
\end{array}
$$

Fig. 5. Temporal patterns P, Q share a k-1 subpattern, joining P and Q yields R.

they share a common (k–1)-pattern on the last k-1 labels. Each temporal pattern describes the desired (k+1)-pattern except for one label. If we join P and Q to the temporal pattern R, as it is illustrated in Fig. 5, then there is only one interval relation missing in R. The missing interval relation (and its inverse) describes the relation between the first labels of P and Q (l_1^p and l_1^q). Now we can extend R to a set of candidates by applying Allen's interval relations. The missing value in R is substituted by each of Allen's interval relations. Hence, the extension of R leads to seven (k+1)-candidate patterns.

In part, even Apriori's candidate pruning scheme, see (b) above, can be transferred to temporal patterns. After all (k+1)-candidates have been generated, we apply the following pruning: Each k-subpattern of a connected (k+1)-candidate pattern has to be contained in the set of frequent k-patterns. Hence, a candidate pattern which contains non-frequent subpatterns can be dismissed without support evaluation. Note that we do not apply this pruning step to unconnected candidate patterns as we know these patterns may contain infrequent subpatterns.

For the actual candidate generation it is necessary to identify all pairs of frequent k-patterns which share a common ($k-1$)-subpattern as their last part[d] (see Fig. 5). An efficient solution to this problem is provided in Ref. 3. The basic idea is to sort

[d]For a convenient abbreviation we call a n-subpattern of P a n-suffix, if the subpattern consists of the last n labels and relations of P (i.e. it is the last part of P). For example in Fig. 5, Q is a k-suffix of R.

the set of frequent k-patterns F_k in a way that all patterns with common $(k-1)$-suffix constitute contiguous blocks in the ordered set of patterns. To sort the set of frequent patterns we use a vector representation for temporal patterns. For a temporal pattern $P = (s, R)$ of size k the vector representation of P is given by:

$$P = (\underbrace{\underbrace{\underbrace{s(k)}_{\text{1-Suffix}},\ s(k-1),\ R[k-1,k]}_{\text{2-Suffix}},\ s(k-2),\ R[k-2,k-1],\ R[k-2,k]}_{\text{3-Suffix}},\ \ldots).$$

For example the vector representation of the pattern in Fig. 2 is $P = (A, B,$ *meets*$, A,$ *overlaps*$,$ *before*$)$. By ordering F_k alphabetically after the patterns vector representation we can guarantee that all patterns with a common $(k-1)$-suffix are next to each other.

Algorithm 2 uses the ordered set of frequent patterns to generate all candidate patterns. Algorithm 2 consists of two main loops. The outer loop (lines 3–20) iterates over all patterns in the ordered list F_k. The inner loop (lines 6–19) passes through all patterns which belong to the same block as the current pattern marked by the

Algorithm 2 Candidate Generation for FSMSet

```
1: procedure CandidateGeneration($F_k$)   ▷ $F_k$: frequent $k$-patterns in ordered vector representation
2:     $C_{k+1} := \emptyset$                 ▷ $C_{k+1}$: candidate patterns of size $k+1$
3:     for $i$:=1 to $|F_k|$ do
4:         currentBlockStart := getBlockStart($F_k$, $i$)
5:         j := currentBlockStart
6:         while currentBlockStart = getBlockStart($F_k$, $j$) and $j \le |F_k|$ do
7:             $R := F_k[i] \cup F_k[j]$    ▷ join of $F_k[i]$ and $F_k[j]$ according to Figure 5
8:             for all rel $\in \{b, m, o, if, c, is, e\}$ do
9:                 $P$ := extend($R$, rel)       ▷ use rel as the missing relation in $R$
10:                if isConnected($P$) then
11:                    if allSubFrequent($P$, $F_k$) then      ▷ apply pruning scheme
12:                        $C_{k+1} := C_{k+1} \cup P$
13:                    end if
14:                else
15:                    $C_{k+1} := C_{k+1} \cup P$
16:                end if
17:            end for
18:            $++j$
19:        end while
20:    end for
21:    return $C_{k+1}$
22: end procedure
```

outer loop. Hence, the combination of both loops iterates over all pairs of patterns which belong to the same block (i.e. share a common $(k-1)$-suffix). This is achieved by using the help function *getBlockStart* (lines 4 and 6). *GetBlockStart* returns the index of the first pattern in F_k which has the same $(k-1)$-suffix as the pattern indexed in the parameters. Each pair of temporal patterns is joined as described in Fig. 5 (line 7). Afterwards all possible relations are used to extend R to candidate patterns (line 9). Finally, we apply the described pruning scheme for connected patterns (lines 10 and 11) in order to reduce the number of generated candidates.

5.2. *Support evaluation*

The second important step is the support evaluation of the candidate patterns.

As already mentioned, the labels of a valid temporal pattern have the same order as their counterparts in an instance would have. Therefore we can find an instance of a temporal pattern in $\mathbb{S}$ by using finite state machines which subsequently take the temporal intervals of an ordered temporal sequence as input.

It is straightforward to derive a finite state machine from a temporal pattern. For each label in the temporal pattern a state is generated. The finite state machine starts in an initial state. The next state is reached if we input a temporal interval that contains the same label as the first label of the temporal pattern. From now on the next states can only be reached if the shown temporal interval carries the same label as the state and its interval relation to all previously accepted temporal intervals is the same as specified in the temporal pattern. If the finite state machine reaches its last state it also reaches its final accepting state.

We can derive the minimal time window in which this particular pattern instance is visible from the set of temporal intervals which has been accepted by the state machine. We know that the time window contains an instance of the pattern but we do not know whether it is a minimal occurrence. Therefore we apply a two step approach. First we will find all instances of a pattern using finite state machines. Second we will filter out all corresponding time windows which are not minimal occurrences.

To find all occurrences of a pattern in an interval sequence we are maintaining a set of finite state machines.[e] At first, the set only contains the finite state machine that is derived from the candidate pattern. Subsequently, each temporal interval from the interval sequence is shown to every finite state machine in the set. If a finite state machine can accept the temporal interval, a copy of the state machine is added to the set. The temporal interval is shown only to one of these two state machines. Hence, there will always be a copy of the initial state machine in the set trying to find an occurrence of the pattern. In this way we can also handle situations in which single state machines do not suffice. Consider the pattern *A meets B* and

[e]The algorithm's name *FSM*Set also originates from the idea of maintaining a set of *f*inite *s*tate *m*achines.

the interval sequence (1, 2, A), (3, 4, A), (4, 5, B). Without using look ahead a single finite state machine would accept the first temporal interval (1, 2, A). This state machine is stuck as it cannot reach its final state because there is no temporal interval which *is-met-by* (1, 2, A). Hence the pattern instance (3, 4, A), (4, 5, B) could not be found by a single state machine. Here this is not a problem because there is a copy of the first state machine which will find the pattern occurrence.

For the second step, we need to find the set of minimal occurrences out of all instances. This can be done by maintaining a list of time windows for each candidate pattern. Each time a finite state machine reaches its final accepting state the minimal time window is checked against the candidate pattern's list of time windows. If the list already contains a subwindow of the window then the window is dismissed. If the current window is a subwindow of windows in the list then those windows are removed and the current window is added to the list. Thus in the end the list contains the set of minimal occurrences of the pattern.

Algorithm 3 FSMSet: Support Evaluation of Candidate Patterns

```
 1: procedure EVALUATESUPPORT($C$, $\mathbb{S}$)        ▷ $C$: candidate patterns, $\mathbb{S}$: interval sequences
 2:     for all $s \in \mathbb{S}$ do
 3:         fsms := createStateMachines($C$)
 4:         for $i := 1 \ldots$ length($s$) do
 5:             $(b, e, l)$ := $i$-th element of $s$
 6:             new-fsms := process(fsms, $(b, e, l)$)
 7:             for all fsm $\in$ new-fsms do
 8:                 if fsm.isFinallyAccepted() then putSupport($s$, fsm)
 9:                 else fsms := fsms $\cup$ {fsm}
10:             end for
11:         end for
12:     end for
13: end procedure
```

Algorithm 3 shows the assembly of the described ideas on a top level. First, we iterate over the set of all interval sequences. For each interval sequence the initial set of final state machines *fsms* is generated by calling the function *createStateMachines* (line 3), i.e. *fsms* contains one state machine for each candidate pattern. Then, we subsequently process all temporal intervals in the current interval sequence. The function *process* (line 6) takes the set of state machines and the current temporal interval as arguments. It returns a set of state machines *new-fsms*. Each element in *new-fsms* is a copy of a state machine in *fsms* but has accepted the current temporal interval. At this step, the set *fsms* stays untouched. We check all new state machines whether they have reached their final acceptance state (lines 7–10). In that case the procedure *putSupport* is used to potentially add the new occurrence to the list of minimal occurrences. Otherwise the state machines are added to the set *fsms* (line 9) to make them available to the next temporal interval in the sequence. The procedure *putSupport* has to maintain one list of minimal occurrences per pattern and interval sequence. Therefore it also takes the current interval sequence as an argument.

In our implementation of FSMSet we did some minor changes to enhance the algorithm's runtime. For example we divided the big set of state machines *fsms* into subsets. Each subset contains all state machines which are expecting the same label in the next accepted temporal interval. Thus we only need to process a subset of all state machines in each iteration. Also we implemented a pruning mechanism in the final state machines. If the current temporal interval would lead to a time window which is longer than a user defined threshold (our required time constraint on a pattern instance) than this state machine is dismissed. In this way we also dismiss state machines which will never reach their accepting state (state machine which are stuck).

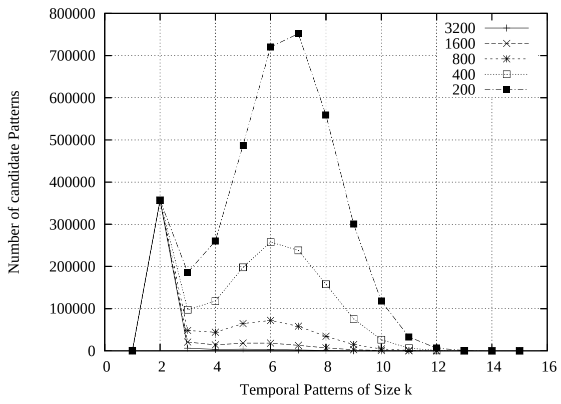
6. Performance Evaluation

In order to evaluate the performance of FSMSet we employed a dataset from our domain. This dataset contains information about the history of 101 250 vehicles. There is one sequence for each vehicle. Each sequence comprises between 14 and 48 temporal intervals. In total, there are 345 different labels and more than 1.4 million temporal intervals in the dataset. A first scan over the data showed that only a subset of Allen's interval relations is present in the dataset, i.e. *before*, *after*, *contains*, *during* and *equals*.

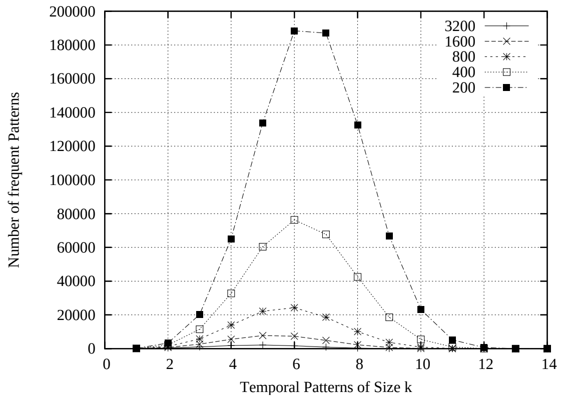
We performed 5 different experiments varying the minimum support threshold from 3 200 down to 200. In each run we measured how many candidates were generated in each iteration of FSMSet and how many of them proved to be frequent during the support evaluation. The runtime of the algorithm was also measured for each experiment. The algorithm is implemented in Java and all experiments were carried out on a SUN Fire X2100 running at 2.2 GHz.

Figure 6(a) shows the number of candidates that are generated in each iteration. Obviously the number of candidates grows rapidly as the minimum support threshold gets lower. This general behaviour is well known from frequent itemset mining. In contrast to the generation of frequent itemsets, Fig. 6(a) shows two distinct peaks. There is one peak for the candidate patterns of size 2 and one peak for patterns of sizes 6–7. Moreover, the first peak does not vary with different minimum support thresholds. This peak is a result of the special candidate generation in the first iteration of FSMSet. The candidates of size 2 are generated by using all the candidates of size 1 (only in subsequent iterations the frequent patterns are used). As the dataset contains 345 different labels and we are using a subset of Allen's interval relations (*before*, *contains* and *equals*) we get $345^2 \cdot 3 = 357075$ candidate patterns of size 2. Consequently the number of 2-candidates is independent of the chosen minimum support threshold.

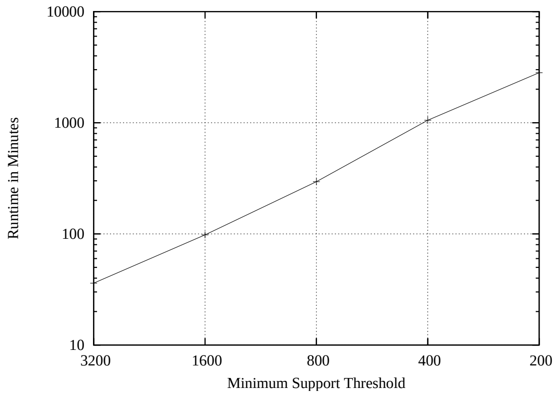
In Fig. 6(b) the number of frequent patterns is depicted for each iteration. For each minimum support threshold the maximum number of frequent patterns is found between the 5-th and 7-th iteration. Again the number of frequent patterns grows rapidly with decreasing minimum support thresholds.



(a)



(b)



(c)

Fig. 6. (a) Candidate patterns generated and (b) frequent patterns found in each iteration for different minimal support thresholds (3200, 1600, 800, 400, 200). (c) Runtime over support threshold.

The increasing number of frequent and candidate patterns also leads to longer runtimes of the algorithm for decreasing minimum support thresholds as Fig. 6(c) shows. While the first experiment (MinSupp = 3200) was finished within 36 minutes subsequent experiments took 98, 295, 1052 and 2 822 minutes.

7. Conclusions

In this paper we presented FSMSet: a new algorithm for discovering frequent temporal patterns. The key advantages of this algorithm are the ability to mine data that consists of several separate interval sequences, a new support definition that allows counting multiple instances of a pattern per sequence, and finally the consideration of time constraints on pattern instances.

Whereas on its own, these features have been described before, e.g. Refs. 7 and 8, an algorithm implementing them all together had not yet been available. As for our and many other applications, combining these features is essential. Therefore our approach opens a broad range of new applications for sequence mining.

Combining the sketched features is far from being straightforward. The main algorithmic challenge is that the downward closure property of support, also known as the Apriori-criteria, is not met. In other words, we had to develop an algorithm that is efficient and still complete with respect to a minimal support threshold, although subpatterns of a frequent pattern may be infrequent. We finally tackled the issue by distinguishing so called connected and unconnected patterns based on the temporal relation *contains*.

Based on an analysis of warranty data from the automotive domain, we showed that FSMSet can be successfully applied to real world data. The results contained valuable knowledge far beyond current approaches and were produced within reasonable time.

References

1. R. Agrawal and R. Srikant, Mining sequential patterns, in *Proc. 11th Int. Conf. Data Engineering* (*ICDE '95*), 1995, pp. 3–14.
2. J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M. Hsu, Freespan: Frequent pattern-projected sequential pattern mining, in *Proc. 6th ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining* (*KDD '00*), 2000, pp. 355–359.
3. H. Mannila, H. Toivonen, and A. I. Verkamo, Discovery of frequent episodes in event sequences, *Data Mining and Knowledge Discovery* **1**(3) (1997) 259–289.
4. N. Méger and C. Rigotti, Constraint-based mining of episode rules and optimal window sizes, in *8th Europ. Conf. Principles and Practice of Knowledge Discovery in Databases* (*PKDD '04*), 2004, pp. 313–324.
5. M. Plantevit, Y. W. Choong, A. Laurent, D. Laurent, and M. Teisseire, M^2SP: Mining sequential patterns among several, dimensions, in *9th Europ. Conf. on Principles and Practice of Knowledge Discovery in Databases* (*PKDD '05*), 2005, pp. 205–216.
6. P.-S. Kam and A. W. Fu, Discovering Temporal Patterns for Interval-Based Events, in 2nd Int. Conf. Data Warehousing and Knowledge Discovery, 2000, pp. 317–326.

7. F. Höppner and F. Klawonn, Finding informative rules in interval sequences, *Intelligent Data Analysis* **6**(3) (2002) 237–255.
8. P. Papapetrou, G. Kollios, S. Sclaroff, and D. Gunopulos, Discovering frequent arrangements of temporal intervals, in *5th IEEE Int. Conf. Data Mining* (*ICDM '05*), 2005.
9. E. Winarko and J. F. Roddick, Discovering richer temporal association rules from interval-based data, in Int. Conf. Data Warehousing and Knowledge Discovery, 2005, pp. 315–325.
10. F. Höppner, Knowledge Discovery from Sequential Data, PhD thesis, TU Braunschweig, 2003.
11. R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and I. Verkamo, Fast discovery of association rules, in *Advances in Knowledge Discovery and Data Mining* (AAAI/MIT Press, 1996), pp. 307–328.
12. J. Ayres, J. Flannick, J. Gehrke, and T. Yiu, Sequential pattern mining using a bitmap representation, in *Proc. 8th ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining* (*KDD '02*), 2002, pp. 429–435.
13. J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu, Prefixspan: Mining sequential patterns by prefix-projected growth, in *Proc. 17th Int. Conf. Data Engineering* (*ICDE '01*), 2001, pp. 215–224.
14. M. J. Zaki, SPADE: An efficient algorithm for mining frequent sequences, *Machine Learning* **42**(1/2) (2001) 31–60.
15. J. F. Allen, Maintaining knowledge about temporal intervals, *Commun. ACM* **26**(11) (1983) 832–843.
16. R. Agrawal, T. Imielinski, and A. Swami, Mining association rules between sets of items in large databases, in *Proc. of the ACM SIGMOD Int. Conf. Management of Data* (*ACM SIGMOD '93*), 1993, pp. 207–216.
17. R. Agrawal and R. Srikant, Fast algorithms for mining association rules, in *Proc. 20th Int. Conf. Very Large Databases* (*VLDB '94*), 1994, pp. 487–499.
18. J. Hipp, U. Güntzer, and G. Nakhaeizadeh, Algorithms for association rule mining – a general survey and comparison, *SIGKDD Explorations* **2**(1) (2000) 58–64.
19. H. Mannila, H. Toivonen, and I. Verkamo, Efficient algorithms for discovering association rules, in *AAAI Workshop on Knowledge Discovery in Databases*, 1994, pp. 181–192.