FSMTree: An Efficient Algorithm for Mining Frequent Temporal Patterns

Steffen Kempe[1], Jochen Hipp[1] and Rudolf Kruse[2]

[1] DaimlerChrysler AG, Group Research, 89081 Ulm, Germany
{Steffen.Kempe, Jochen.Hipp}@daimlerchrysler.com
[2] Dept. of Knowledge Processing and Language Engineering,
University of Magdeburg, 39106 Magdeburg, Germany
Kruse@iws.cs.uni-magdeburg.de

Abstract. Research in the field of knowledge discovery from temporal data recently focused on a new type of data: interval sequences. In contrast to event sequences interval sequences contain labeled events with a temporal extension. Mining frequent temporal patterns from interval sequences proved to be a valuable tool for generating knowledge in the automotive business. In this paper we propose a new algorithm for mining frequent temporal patterns from interval sequences: *FSMTree*. *FSMTree* uses a prefix tree data structure to efficiently organize all finite state machines and therefore dramatically reduces execution times. We demonstrate the algorithm's performance on field data from the automotive business.

1 Introduction

Mining sequences from temporal data is a well known data mining task which gained much attention in the past (e.g. Agrawal and Srikant (1995), Mannila et al. (1997), or Pei et al. (2001)). In all these approaches, the temporal data is considered to consist of events. Each event has a label and a timestamp. In the following, however, we 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.

At DaimlerChrysler we are interested in mining interval sequences in order to further extend the knowledge about our products. Thus, in our domain one interval sequence may describe 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 may describe stopovers in a garage or the installation of additional equipment. Hence, mining these interval sequences might help us in tasks like quality monitoring or improving customer satisfaction.

2 Foundations and Related Work

As mentioned above we represent a temporal interval as a triplet (b, e, l).

Definition 1. *(Temporal Interval) Given a set of labels $l \in L$, 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 $(b_1, e_1, l_1), (b_2, e_2, l_2), \ldots, (b_n, e_n, l_n) \in I$ is an interval sequence, if*

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

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

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

Equation 1 above is referred to as the *maximality assumption* (Höppner (2002)). 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 beginning (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 (or 13 if one includes inverse relations). These interval relations have been described in Allen (1983) and are depicted in Figure 1. Each relation of Figure 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" or "n-pattern".*

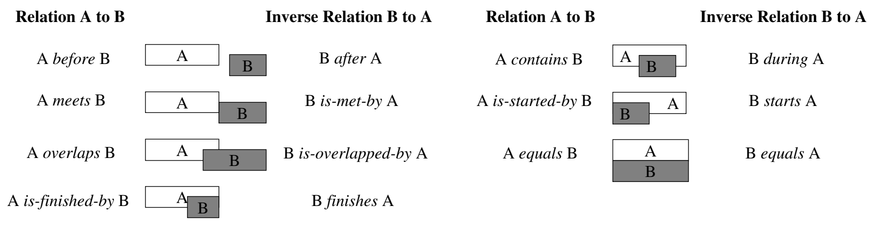
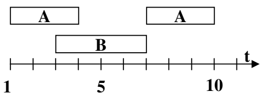


Fig. 1. Allen's Interval Relations

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.)

Figure 2.a shows an example of an interval sequence. The corresponding temporal pattern is given in Figure 2.b.

Note that a temporal pattern is not necessarily valid in the sense that it must be possible to construct an interval sequence for which the pattern holds true. On the 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) An interval sequence $S = (b_i, e_i, l_i)_{1 \leq i \leq n}$ conforms to a n-pattern $P = (s, R)$, 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. Next, we define the support of a temporal pattern.

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.*

Definition 6. *(Support) The support of a temporal pattern P for a given set of interval sequences $\mathbb{S}$ is given by the number of minimal occurrences of P in $\mathbb{S}$: $Sup_{\mathbb{S}}(P) = |\{[b, e] : [b, e]$ is a minimal occurrence of P in $S \wedge S \in \mathbb{S}\}|$.*

As an illustration consider the pattern A *before* A in the example of Figure 2.a. The time window $[1, 11]$ is not a minimal occurrence as the pattern is also visible e.g. in its subwindow $[2, 9]$. Also the time window $[5, 8]$ is not a minimal occurrence. It does not contain an instance of the pattern. The only minimal occurrence is $[4, 7]$ as the end of the first and the beginning of the second A are just inside the time window.

The mining task is to find all temporal patterns in a set of interval sequences which satisfy a defined minimum support threshold. Note that this task is closely related to frequent itemset mining, e.g. Agrawal et al. (1993).

Previous investigations on discovering frequent patterns from sequences of temporal intervals include the work of Höppner (2002), Kam and Fu (2000), Papapetrou et al. (2005), and Winarko and Roddick (2005). 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*. It can be interpreted as the probability to see an instance of the pattern within the 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.

For our application neither of the support definitions turned out to be satisfying. Höppner's temporal support of a pattern is hard to interpret in our domain, as it is generally not related to the number of instances of this pattern in the data. Also neglecting multiple instances of a pattern within one interval sequence is inapplicable when mining the repair history of vehicles. Therefore we extended the approach of minimal occurrences in Mannila (1997) to the demands of temporal intervals. In contrast to previous approaches, our support definition allows (1.) to count the number of pattern instances, (2.) to handle multiple instances of a pattern within one interval sequence, and (3.) to apply time constraints on a pattern instance.

3 Algorithms FSMSet and FSMTree

In Kempe and Hipp (2006) we presented *FSMSet*, an algorithm to find all frequent patterns within a set of interval sequences $\mathbb{S}$. The main idea is to generate all frequent temporal patterns by applying the Apriori scheme of candidate generation and support evaluation. Therefore *FSMSet* consists of two steps: generation of candidate sets and support evaluation of these candidates. These two steps are alternately repeated until no more candidates are generated. The Apriori scheme starts with the frequent 1-patterns and then successively derives all k-candidates from the set of all frequent (k-1)-patterns.

In this paper we will focus on the support evaluation of the candidate patterns, as it is the most time consuming part of the algorithm. *FSMSet* uses finite state machines which subsequently take the temporal intervals of an interval sequence as input to find all instances of a candidate pattern.

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

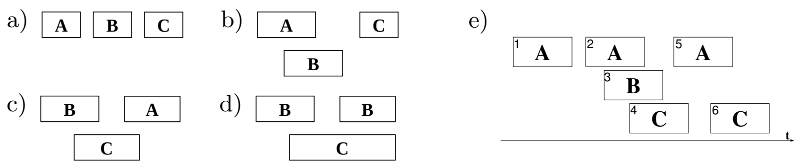


Fig. 3. a) – d) four candidate patterns of size 3 e) an interval sequence

	1	2	3	4	5	6
$S_a()$	$S_a(1)$	$S_a(2)$	$S_c(3)$	$S_c(3,4)$	$S_a(5)$	$\mathbf{S_a(1,3,6)}$
$S_b()$	$S_b(1)$	$S_b(2)$	$S_d(3)$	$S_d(3,4)$	$S_b(5)$	$\mathbf{S_b(2,3,6)}$
$S_c()$			$S_a(1,3)$		$\mathbf{S_c(3,4,5)}$	
$S_d()$			$S_b(2,3)$			

Table 1. Set of state machines of *FSMSet* for the example of Figure 3. Each column shows the new state machines that have been added by *FSMSet*.

its final accepting state. Consequently the temporal intervals that have been accepted by the state machine are an instance of the temporal pattern.

The minimal time window in which this pattern instance is visible can be derived from the temporal intervals which have been accepted by the state machine. We know that the time window contains an instance but we do not know whether it is a minimal occurrence. Therefore *FSMSet* applies a two step approach. First it will find all instances of a pattern using state machines. Then it prunes all time windows which are not minimal occurrences.

To find all instances of a pattern in an interval sequence *FSMSet* is maintaining a set of finite state machines. At first, the set only contains the state machine that is derived from the candidate pattern. Subsequently, each temporal interval from the interval sequence is shown to every state machine in the set. If a 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 a new instance of the pattern. In this way *FSMSet* also can handle situations in which single state machines do not suffice. Consider the pattern *A meets B* and 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 instance.

Figure 3 and Table 1 give an example of *FSMSet*'s support evaluation. There are four candidate patterns (Figure 3.a – 3.d) for which the support has to be evaluated on the given interval sequence in Figure 3.e. At first, a state machine is derived for each candidate pattern. The first column in Table 1 corresponds to this initialization (state machines $S_a - S_d$). Afterwards each temporal interval of the sequence is used as input for the state machines. The first temporal interval has label A and can only be accepted by the state machines $S_a()$ and $S_b()$. Thus the new state machines $S_a(1)$ and $S_b(1)$ are added. The numbers in brackets refer to the temporal intervals of the interval sequence that have been accepted by the state machine. The second temporal interval carries again the label A and can only be accepted by $S_a()$ and $S_b()$. The third temporal interval has label B and can be accepted by $S_c()$ and $S_d()$. It also stands to the first A in the relation *after* and to the second A in the relation *is-overlapped-by*. Hence also the state machines $S_a(1)$ and $S_b(2)$ can accept this interval. Table 1 shows all new state machines for each temporal interval of the interval sequence. For this example the approach of *FSMSet* needs 19 state machines to find all three instances of the candidate patterns.

A closer examination of the state machines in Table 1 reveals that many state machines show a similar behavior. E.g. both state machines S_c and S_d accept exactly the same temporal intervals until the fourth iteration of *FSMSet*. Only the fifth temporal interval cannot be accepted by S_d. The reason is that both state machines share the common subpattern *B overlaps C* as their first part (i.e. a common prefix pattern). Only after this prefix pattern is processed their behavior can differ. Thus we can minimize the algorithmic costs of *FSMSet* by combining all state machines that share a common prefix. Combining all state machines of Figure 3 in a single data structure leads to the prefix tree in Figure 4. Each path of the tree is a state machine. But now different state machines can share states, if their candidate patterns share a common pattern prefix. By using the new data structure we derive a new algorithm for the support evaluation of candidate patterns — *FSMTree*.

Instead of maintaining a list of state machines *FSMTree* maintains a list of nodes from the prefix tree. In the first step the list only contains the root node of the tree. Afterwards all temporal intervals of the interval sequence are

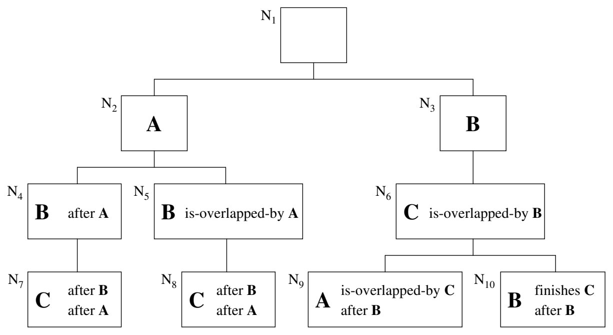


Fig. 4. FSMTree: prefix tree of state machines based on the candidates of Figure 3

	1	2	3	4	5	6
$N_1()$	$N_2(1)$	$N_2(2)$	$N_3(3)$	$N_6(3,4)$	$N_2(5)$	$\mathbf{N_7(1,3,6)}$
			$N_4(1,3)$		$\mathbf{N_9(3,4,5)}$	$\mathbf{N_8(2,3,6)}$
			$N_5(2,3)$			

Table 2. Set of nodes of *FSMTree* for the example of Figure 3. Each column gives the new nodes that have been added by *FSMTree*.

processed subsequently. Each time a node of the set can accept the current temporal interval its corresponding child node is added to the set. Table 2 shows the new nodes that are added in each step if we apply the prefix tree of Figure 4 to the example of Figure 3. Obviously the algorithmic overhead is reduced significantly. Instead of 19 state machines *FSMTree* only needs 11 nodes to find all pattern instances.

4 Performance Evaluation and Conclusions

In order to evaluate the performance of FSMTree in a real application scenario 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 about 1.4 million temporal intervals in the dataset.

We performed 5 different experiments varying the minimum support threshold from 3 200 down to 200. For each experiment we measured the runtimes of *FSMSet* and *FSMTree*. The algorithms are implemented in Java and all experiments were carried out on a SUN Fire X2100 running at 2.2 GHz. Figure 5 shows that *FSMTree* clearly outperforms *FSMSet*. In the first experiment *FSMTree* reduced the runtime from 36 to 5 minutes. The difference

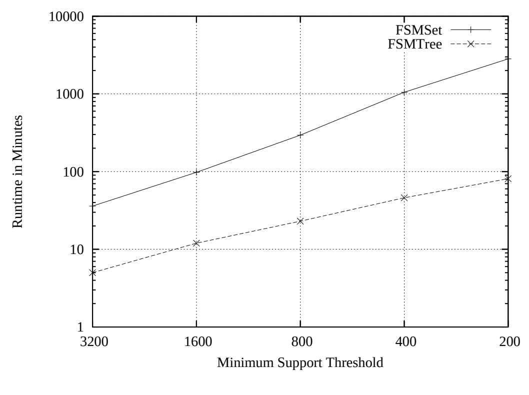


Fig. 5. Runtimes of *FSMSet* and *FSMTree* for different support thresholds.

between *FSMSet* and *FSMTree* even grows as the minimum support threshold gets lower. For the last experiment *FSMSet* needed two days while it took *FSMTree* only 81 minutes. The reason for *FSMTree*'s huge runtime advantage at low support threshold is that as the support threshold decreases the number of frequent patterns increases. Consequently the number of candidate patterns increases too. The number of candidates is the same for *FSMSet* and *FSMTree* but *FSMTree* combines all patterns with common prefix patterns. If there are more candidate patterns the chance for common prefixes increases. Therefore *FSMTree*'s ability to reduce the runtime will increase (compared to *FSMSet*) as the support threshold gets lower.

In this paper we presented *FSMTree*: a new algorithm for mining frequent temporal patterns from interval sequences. *FSMTree* is based on the Apriori approach of candidate generation and support evaluation. For each candidate pattern a finite state machine is derived to parse the input data for instances of this pattern. *FSMTree* uses a prefixtree-like data structure to efficiently organize all finite state machines. In our application of mining the repair history of vehicles *FSMTree* was able to dramatically reduce execution times.

References

AGRAWAL, R., IMIELINSKI, T. and SWAMI, A. (1993): Mining association rules between sets of items in large databases. In: *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (ACM SIGMOD '93).* 207–216.

AGRAWAL, R. and SRIKANT, R. (1995): Mining sequential patterns. In: *Proc. of the 11th Int. Conf. on Data Engineering (ICDE '95).* 3–14.

ALLEN, J. F. (1983): Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843.

HÖPPNER, F. and KLAWONN, F. (2002): Finding informative rules in interval sequences. *Intelligent Data Analysis*, 6(3):237–255.

KAM, P.-S. and FU, A. W.-C. (2000): Discovering Temporal Patterns for Interval-Based Events. In: *Data Warehousing and Knowledge Discovery, 2nd Int. Conf., DaWaK 2000.* Springer, 317–326.

KEMPE, S. and HIPP, J. (2006): Mining Sequences of Temporal Intervals. In: *10th Europ. Conf. on Principles and Practice of Knowledge Discovery in Databases* Springer, Berlin-Heidelberg, 569–576.

MANNILA, H., TOIVONNEN, H. and VERKAMO, I. (1997): Discovery of frequent episodes in event sequences. *Data Mining and Knowl. Discovery*, 1(3):259–289.

PAPAPETROU, P., KOLLIOS, G., SCLAROFF, S. and GUNOPULOS, D. (2005): Discovering frequent arrangements of temporal intervals. In: *5th IEEE Int. Conf. on Data Mining (ICDM '05).* 354–361.

PEI, J., HAN, J., MORTAZAVI, B., PINTO, H., CHEN, Q., DAYAL, U. and HSU, M. (2001): Prefixspan: Mining sequential patterns by prefix-projected growth. In: *Proc. of the 17th Int. Conf. on Data Engineering (ICDE '01).* 215–224.

WINARKO, E. and RODDICK, J. F. (2005): Discovering Richer Temporal Association Rules from Interval-Based Data. In: *Data Warehousing and Knowledge Discovery, 7th Int. Conf., DaWaK 2005.* Springer, Berlin-Heidelberg, 315–325.