Tree Growth based Episode Mining without Candidate Generation

M. Baumgarten
Faculty of Informatics, University of Ulster, Newtownabbey, BT37 0QB, UK

A.G. Büchner
Faculty of Informatics, University of Ulster, Newtownabbey, BT37 0QB, UK

J.G. Hughes
University of Ulster, Coleraine, BT52 1SA, UK

Mining for frequent episodes has been an active research area in recent years. Numerous algorithms have been developed to discover different types of episodes, where most of them adopt an a priori-like approach that generates candidates and then recognises these candidates to determine their support. However, such methods are computationally expensive, depending on the size and structure of the input data. Within this paper a tree growth based method is presented discovering episodes without candidate generation. The presented method only consists of a recognition phase that dynamically extends a specialised tree structure to efficiently store and process episodes.

Keywords: data mining methods, pattern discovery, episodes mining

1 Introduction

Mining large databases to discover different types of patterns has been a challenge for a vast number of researchers from various domains. Two of the most popular pattern types are associations and sequences. Different techniques have been introduced over the past decade to discover such patterns and to utilize them for different domains. A third and important type of pattern incorporating associative as well as sequential structures is known as episodes. An episode is defined as a collection of events, following a certain structure, which are relatively close to each other in time based on a given threshold. Unlike associations and sequences, episodes are discovered using a single set structure also called an event sequence. Such an event sequence is a sequence of items, where each item has an associated time of occurrence.

Past episode discovery approaches are mostly based on traditional a priori-style algorithms that generate candidates to build episodes in an iterative fashion. Such methods usually generate candidates of length n (Phase 1) and then determine their occurrence through a recognition step (Phase 2). Due to the problems that are associated with each phase, traditional candidate generation methods can be very costly. However, since Phase 2 depends on the input of Phase 1, the problem is aggravated even further, because the number of candidates C_k generated by Phase 1 can be potentially very large. Phase 2, taking C_k as an input, needs to scan the database to determine how often each candidate $c_j \in C_k$ exists within a given event sequence. Thus, as C_k grows, the run time of Phase 2 increases as well.

The approach presented within this paper eliminates the need to generate candidates, growing episodes in an incremental fashion including only k-episode patterns that occur within a given event sequence (k representing the size of an episode). Essentially, the proposed technique only consists of a recognition phase, extending dynamically a structural representation of all k-episodes, found at a given state of the discovery process and adapting their frequency.

The paper is organized as follows. In Section 2, related work is reviewed and drawbacks are shown. Within Section 3 episode terminologies are defined. Section 4, which is the heart of this paper, describes the episode detection algorithm. In Section 5, the search space and the algorithm complexity are analysed. In Section 6 some experiments are performed and evaluated with respect to the overall performance, before Section 7 concludes the paper and outlines future work.

2 Related Work

Episodes are a special kind of pattern which, by definition, occur close to each other in time. Such patterns provide a powerful technique to analyze time series related data, such as error and status log files or behavioural patterns, which contain related items or in this case *events*. Examples are found in the telecommunications sector, fraud detection applications or stock market analyses.

In [4] and [5] an episode is defined as an a collection of events, following a certain structure, that are relatively close to each other in time based on a

given threshold. The WINEPI algorithm presented in this paper uses an iterative candidate generation method to discover serial and parallel episodes. With this method candidates from previous iterations are used to generate a new seed of candidates of size $n + 1$ for the next iteration. This method is devised to consist of two phases: candidate *generation* phase and candidate *recognition* phase. As the name suggest, the former generates candidates, while the latter determines if a given candidate fulfils a given minimum support constraint. Although the method's flexibility and powerfulness, it suffers from two main drawbacks. Firstly, candidate-based methods tend to generate too many candidates, specifically during its early iterations, and secondly, a complete database scan is required for each candidate generation phase.

In [3], two methods (slice scan and selective hash) are introduced to address the problems mentioned above. Simplified, slice scan generates a collection of candidates C_k's, which is defined as a slice containing candidates of size n to n + *slice size* (S_l). Thereafter, the database is scanned for each slice to reduce the number of data base scans. This is an effective technique since it reduces the number of database scans by a factor $S_l - 1$. However the cost for the reduction in database scans is considerable. Firstly, a higher number of candidates need to be tested and secondly, the number of candidates generated during early iterations is even larger than [5]. To reduce the number of candidates a method called selective hash is introduced that uses an item hashing technique to filter out rare candidate 2-episodes, which reduces the number of candidates. [3] provides a significant performance improvement over WINEPI, however, it still suffers the problems that come from using a candidate generation approach.

[2] introduces a method called *frequent pattern growth* that effectively mines patterns without candidate generation. This is done by utilizing methods that preserve the essential grouping of original data elements that are used for mining the desired patterns. The analysis phase then focuses on counting the occurrence of the relevant data sets. A divide-and-conquer methodology is further introduced to reduce the search space through the partitioning of the original data set. The presented method is similar to the one outlined in this paper. However, the frequent pattern growth approach uses only a single tree structure and is therefore not optimised for episode mining.

3 Episode Fundamentals

Like most existing episode-related research, our algorithm uses the notation presented in [4]. For better understanding and for the sake of completeness relevant definitions are given within this section.

3.1 Event-Related Constructs

Events: Let E be a class of elementary event types, then an event can be defined as a pair $p = (e, t)$, where $e \in E$ and t is defined as a time related statement. An example of an event is p_n = (A, December:5:2002:22:34:11).

Event Sequence: An event sequence S is defined as the triple $S = (t_S, t^S, S')$, where t_S, is the starting time and t^S the closing time of the event sequence and S' contains an ordered list of events. So that S' is defined as

$$S = (t_1, t_n, [(e_1, t_1), (e_2\ t_2, (e_3\ t_3, \ldots, (e_n\ t_n)]),$$

where each $e \in E$, each $t_S \le t_i \le t^S$ and each $t_i < t_{i+1}$. The period β covered by S is then $\beta = t^S - t_S$. An example event sequence is shown in Figure 1, where the events are shown on the top of the time line and the associated time of occurrence is shown on the bottom.

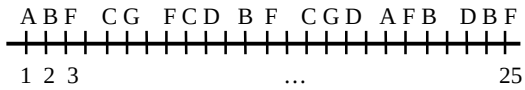


Figure 1: Example Event Sequence

Event Window: An event window W on S' is an event sequence and defined as $W = (t_W, t^W, S'')$, where $t_W < t^W$ and $t_S \le t_W$ and S'' contains all $p \in S'$, where $t_W \le t_i \le t^W$. The width of an event window w is then defined as $w = t^W - t_W$. Figure 2 shows an example event sequence where the event window of width 3 is marked; it further visualizes the shifting (see windowing model) across the event sequence.

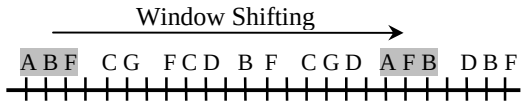


Figure 2: Event Sequence, $w = 3$

Event Window Increment[1]**:** The window increment I defines how much the event window W is shifted along S' and is between $0 \le I \le w$. The set of all windows in S is denoted by V where the size V^N of V depends on the time period covered, the window increment, and the window width and is defined as

$$V^N = \left[\frac{(t_1 - t_n) - w}{I} \right] + 1.$$

3.2 Episodes

An episode is defined as a collection of events, following a certain structure, that are relatively close to

[1] Other increment methods such as "every event" are also possible.

each other in time based on a given threshold. Figure 3 shows different types of episodes that are described next. Episodes are denoted by Φ and |= is used to denote that an episode occurs within a given event sequence.

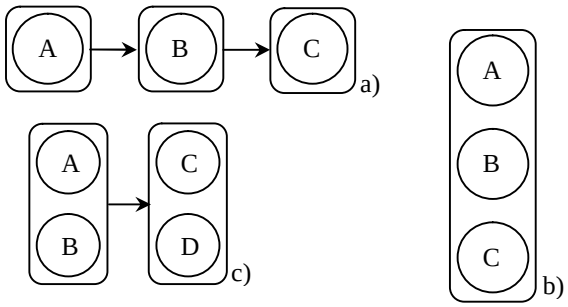


Figure 3: Episode Types

Serial Episodes, Φ_S (a)**:** are defined as an ordered list (sequence) of events that occur within a given event sequence relatively close to each other in time. Such that (A,B) ≠ (B,A). Figure 3 (a) shows an example episode where A precedes B, and B precedes C.

Parallel Episodes, Φ_P (b): are defined as a set (association) of events that occur within a given event sequence relatively close to each other. Such that (A,B) = (B,A). Figure 3 (b) shows and example episode where A, B and C occur independently of their order.

Composite Episodes, Φ_C (c): can be seen as a superset of parallel and serial episodes (Figure 4). Effectively there are a combination of serial and parallel episodes in a way that they are defined as an ordered list (serial episodes) of sets (parallel episodes) of events that occur within a given event sequence relatively close to each other in time. Composite episodes can be built, by concatenating parallel episodes in a serial fashion. Figure 3 (c) shows an example episode where (A, B) precedes (C, D). There are no constraints on the order of (A, B) or (C, D).

Frequent Episodes: An episode is called frequent if it occurs frequently enough, based on a user's threshold, within a given event sequence. The number of occurrences *o* is based on how often an episode occurs in all event windows. Its frequency is calculated as $f = o / V^N$. Due to the fact that the underlying structure of episodes is set-based, multiple occurrences of an episode in a given event window *W* is counted as single occurrence.

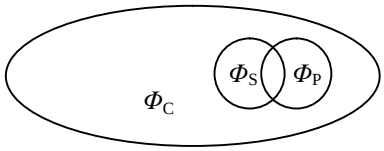


Figure 4: Composite Episodes = $\Phi_C \cup \Phi_S \cup \Phi_P$

3.3 Windowing Model

As described earlier episodes are defined as collections of events, which are close to each other, where close is defined by a given threshold. This threshold forms a window that, by sliding it over the event sequence *S'*, provides a number of event sub-sequences *S''*. This creates a view on the event sequence splitting it into sub-event sequences containing only events that are close enough to each other in time and thus simplifying the discovery process.

ID	*S''*	ID	*S''*	ID	*S''*
1	ABF	9	CD	17	AF
2	BF	10	DB	18	AFB
3	FC	11	BF	19	FB
4	CG	12	FC	20	BD
5	CG	13	CG	21	DB
6	GF	14	CGD	22	DBF
7	FC	15	GD		
8	FCD	16	DA		

Table 1: Events (Window Width: 3)

Table 1 shows a collection of all possible event windows of the event sequence shown in Figure 2, where the window width is set to 3 and the window increment is set to 1.

Although this technique is effective in that it splits up the event sequence and creates a virtual view thereof without the need to alter the original event sequence, it suffers a certain drawback. A shift of an event window does not necessarily result in a change of content of the shifting window. Thus, at least one event drops out or enters in to the scope of the window. To avoid re-processing the same window (the same content), a second window is used that virtually looks ahead of the current window and determines how many shifts are required until the content of the current window changes. This offset is then used as an increment value to update the frequency of all episodes contained in the current window. The next shift repositions the window to this offset position ensuring that the new window differs from the old one. This reduces the number of window shifts V^S and thus run time. V^S partially depends on *w* and *I* but mainly on the event distribution within the event sequence. For a worst case scenario the number of shifts required is $V^S = V^N$. However, for realistic scenarios $V^S << V^N$.

4 Episode Detection Algorithm

Given an event sequence S, an event window W of width w, an event window increment value I and a minimum frequency value $f_{\min}$, the goal is to find all composite episodes contained in S that satisfy $f_{\min}$ Within this section a tree growth based algorithm is introduced to efficiently discover composite episodes.

Taking advantage of the fact that episodes can be described as directed acyclic graphs, a tree structure T is introduced combining sub-episodes into a single branch and therefore minimising memory usage and processing efforts. For instance, given two episodes (A,B,C) and (A,B,D), standard techniques store these episodes separately resulting in the need to store A and B twice. Using a tree like structure the two episodes are combined into (A,B,(C|D)), without loosing generality. T^P will be used to denote the structure that holds parallel episodes and T^S will be used to store the serial combinations representing composite episodes.

In order to discover composite episodes effectively the proposed algorithm is split into two phases. First, D^{P} (lines 2 to 11 of Algorithm 1) is designed to efficiently discover parallel episodes. Second, the result of D^{P} is used by $D^{\mathrm{S|C}}$ (lines 12 to 20) to discover all composite episodes by building a serial concatenation of all parallel episodes.

4.1 Building Parallel Episodes

Building parallel episodes is similar to the problem of finding associative patterns [1] within a given set of items (events). This is due to the fact that parallel episodes contain a set of events and thus the original order in S'' can be ignored. D^P extends iteratively the depth of T^P alternating between building and pruning phases that first build a new level on T^P, thus discovering all episodes of *size* + 1 and updates relevant occurrence values. After the entire event sequence has been scanned, all nodes that do not support $f_{\min.}$ are removed and branches that can no longer be extended are disabled. This process continues until the root node of the tree is disabled and therefore T^P is marked as inactive. Note, that the root node of T^P is a dummy, not containing any value. To extend the tree a sorted S''_k is used and its values are recursively overlaid onto the current tree structure. If there is a set of nodes forming a pattern on T^P of size i, then the leaf node is extended with all remaining events in S''_k forming episodes of size $i + 1$. Reoccurring patterns within the same S''_k are discarded. Figure 5(a) shows a fully deployed tree based on the two windows highlighted in Figure 2. Note, that their exists no pattern (F,B) because this is covered by (B,F) since (F,B) = (B,F).

à $V(S)$, $f_{\min.}$

D^{P}	1)	$i := 0$;
	2)	**while** (T^P is active) **do**
	3)	**while**(V.hasNext) **do**
	4)	sort S''_k
	5)	extend T^P_{i+1} with ($S''_k \mid= \Phi_{\mathrm{p}}$) of size *i +1* and set *frequency* f_Φ
	6)	**od;**
	7)	remove all nodes of T^P_{i+1} where $f_\Phi < f_{\min.}$ and disable non-extendable branches
	8)	$i := i + 1$;
	9)	**od;**
	10)	// T_P is fully deployed containing all Φ_{p}
$D^{\mathrm{S\|C}}$	11)	$i := 0$;
	12)	**while** (T^S is active) **do**
	13)	**while**(V.hasNext) **do**
	14)	$R_k = T^P (S''_k \mid= \Phi_{\mathrm{p}})$
	15)	extend T^S_{i+1} with ($R_k \mid= \Phi_C$) of size *i +1* and set *frequency* f_Φ
	16)	**od;**
	17)	remove all nodes of T^S_{i+1} where $f_\Phi < f_{\min.}$ and disable non-extendable branches
	18)	$i := i + 1$;
	19)	**od;**

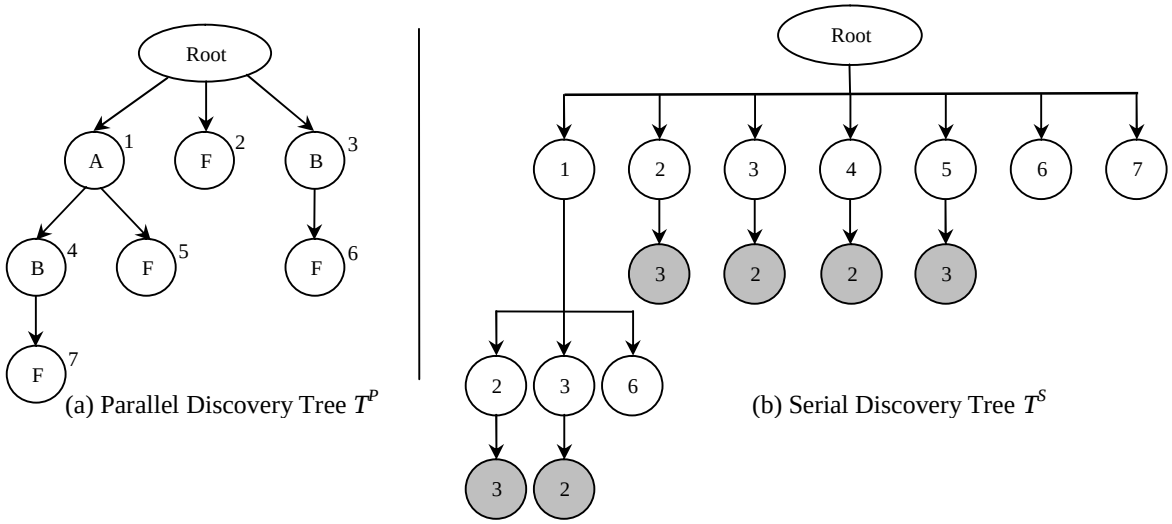
ß T^S containing all Φ

Algorithm 1: Main Algorithm

4.2 Building Composite Episodes

As described in section 3.2, building composite episodes follows the problem of finding sequential patterns. Thus, $D^{S|C}$ uses the parallel episodes contained in T^P to deploy T^S. This requires the re-discovery of all parallel episodes that are contained in a given S''_k, resulting in $R_k = (r_1, r_2, r_3, \dots)$, where all elements in R_k are defined by the triple (t_Φ, t^Φ, Φ_{p}), where t_Φ represents the start time, t^Φ the end time and Φ_{p} the parallel episode per se.

$D^{S|C}$, similar to D^P, also works iteratively and alternates between the building and pruning phase. The disabling or deactivating of nodes is performed identically to its parallel counterpart. Figure 5(b) shows a fully deployed T^S based on T^P, where marked nodes do not occur at least twice. Nodes in T^S do not contain any events directly; instead they store references to nodes in T^P, indicated through the numerical value attached to each node in T^P and used for T^S. Thus, composite episode patterns are represented in the structure of T^S using references to T^P. For instance, pattern (1,6) contained in T^S reflects the episode (A(B,F)). Updating the tree structure to extend the depth to include new episodes of size $i + 1$ is similar to the update process for parallel episodes. $D^{S|C}$ uses a set of all Φ_{p} contained in a given S''_k to update T^S.



(a) Parallel Discovery Tree T^P (b) Serial Discovery Tree T^S

Figure 5: Example Tree Structures

To extend T^S, all $r \in R_k$ are overlaid recursively onto T^S. If a set of nodes is found that forms an episode pattern of size i then the leaf node is extended with all remaining r_j for which $t^{\Phi}(r_i) < t_{\Phi}(r_{i+n})$.

4.3 Optimisation

Algorithm 1 is not optimised to discover exclusively serial or parallel episodes. However, it can be constrained to do so. By limiting the depth of T^S to one, only parallel episodes are discovered because T^S only holds episodes containing a single set of events. For serial episodes the depth of T^P needs to be set to one forcing $D^{S|C}$ to deploy itself with nodes that contain only single events. While this shows the flexibility of the proposed architecture, it also highlights the inefficiency of the procedure when only serial or parallel episodes are desired, since only one tree is required to build either parallel or serial episodes. Separating D^P into a single algorithm discovering just parallel episodes avoids the construction of T^S and is therefore more efficient. For serial episodes, D^P can be used as well simply by excluding the sorting method shown in line 5 and therefore using the original S' directly.

5 Search Space and Complexity

5.1 Search Space

The search space Ξ for episodes depends on the number of distinct events and the selected window size, which limits the maximum number of events in any given window and therefore the maximum size of an episode. To analyse the search space it is assumed that w is greater then the length of the input event sequence such that $t_w = t_S$ and $t^W > t^S$. Thus, there is only a single event window containing an event sub-sequence S'' of size k. It is also assumed that there are no two events of the same type. The number of serial or parallel episodes of a certain size m are identical and calculated as $\Xi_{P|S}(m) = \binom{k}{m}$. The overall number of serial or parallel episodes is calculated as $\Xi_{P|S}(k) = 2^k - 1$. The search space for composite episodes is calculated as

$$\Xi_C(k) = \sum_{j=0}^{k-1} 3^j = \begin{cases} 1 & if\ k = 1 \\ \Xi_C(k-1) * 3 + 1 & if\ k > 1 \end{cases}.$$

This shows that the search space for composite episodes exceeds the search space for parallel or serial episodes significantly. This is due to the fact that composite episodes are build upon parallel episodes, thus if the number of parallel episodes increases the number of possible serial combinations increases in an exponential fashion.

5.2 Complexity

To analyse the complexity O of proposed approach, an event sequence is used covering a time period β, in which an event takes place every second and I is set to 1. The number of windows to be processed for each scan over the database is then $V^N = m - w + 1$. For non-artificial data V^N can be replaced with V^S since not every window necessarily needs to be processed. However, for this analysis $V^N = V^S$.

The number of database scans, Ω, required depends on w since it defines the size of the event window and therefore the maximum number of events in any given W. For this scenario the number of events is $k = w + 1$. As outlined in Section 4, D^P and $D^{S|C}$ need

both in the worst case scenario $\Omega = k + 1$ database scans. Assuming also that it takes time α to discover all episodes contained in S, then the complexity for D^P is $Q_{D^P} = \Omega V^N + a(f_P \models S)$. The overall complexity for composite episodes is $Q_{D^{S|C}} = \Omega V^N + a(f_C \models S) + Q_{D^P}$.

6 Experimental Results

To evaluate the proposed algorithm several experiments have been performed on different artificial data sets. The events within all data sets are evenly distributed following a uniform distribution. All methods have been implemented in Java and the experiments have been run on a PC 450 MHz Pentium 3 Processor with 256 MB RAM and Windows 2000 as operating system.

size	$t(s)_P$	Φ_P	$t(s)_C$	Φ_C
1	<1	14	<1	16383
2	<1	91	4	98305
3	<1	364	6	274431
4	<1	1001	11	471041
5	<1	2002	7	553983
6	<1	3003	6	471041
7	<1	3432	4	297727
8	<1	3003	2	141569
9	<1	2002	1	50623
10	<1	1001	1	13441
11	<1	364	<1	2575
12	<1	91	<1	337
13	<1	14	<1	27
14	<1	1	<1	1
$\sum \Phi_{P\|C}$		16383		2391484

Table 2: Tree Deployment Characteristics

6.1 Tree Deployment

To analyse how effective the tree structure can be deployed independent from other factors (such as the number of windows) a data base is created containing 14 distinct events, where each event occurs 1 second after its predecessor. The window with w is set to 20 seconds so that there is just one window. Thus eliminating V^S, the time required to scan the data base can be neglected due to the small size of the data set. The minimum frequency is set to a single occurrence, which means that all episodes will be discovered. Table 2 shows the time taken to create each level and the number of episodes added at each level for both parallel (Φ_P) and composite (Φ_C) episodes. The sums of all nodes are shown in the last row, validating the equations given in the previous section.

6.2 Performance

In order to analyse the performance of the episode detection algorithm, a data base has been created containing 300 distinct events types. The overall length of the event sequence is 2,000 events covering a period of approximately 24 hours. In Table 3 all parallel and composite episodes have been discovered using a fixed $f_{min.} = 0.03\%$ and a varying window width w. Execution times are between less than one second and 30 minutes, depending on the window size. For a larger w the number of episodes for both parallel and composite episodes increases rapidly, indicating the homogeneous distribution of events within the database. Table 4 shows the behaviour for a fixed $w = 100$ and varying $f_{min.}$., resulting in execution times between 5 and 66 seconds in which up to 36,000 episodes are discovered. Both Table 3 and Table 4 show a robust and efficient behaviour even if larger numbers of episodes are discovered.

w(s)	t(s)	$\|\Phi_P\|$	t(s)	$\|\Phi_C\|$
10	<0	322	<0	322
20	<0	325	<0	326
40	1	859	1	1456
80	3	3375	12	8893
160	26	13264	214	68420
200	147	42228	1898	427531

Table 3: Varying w(s) and fixed $f_{min.} = 0.03\%$

$f_{min.}$	t(s)	$\|\Phi_P\|$	t(s)	$\|\Phi_C\|$
0.1	3	857	5	1427
0.05	5	3553	25	9745
0.008	8	8022	59	30904
0.004	9	8619	62	34198
0.002	9	8813	65	35397
0.001	9	8940	66	36135

Table 4: Varying f_{min} and fixed w(s) = 100

6.3 Scale Up

To analyse the scale up properties, different data sets have been evaluated scaling different parameters that are of influence.

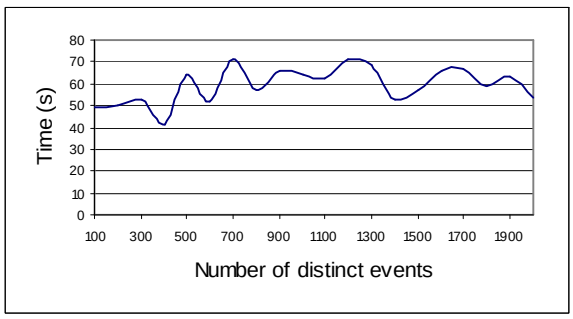


Figure 6: Runtime Characteristics, 2,000 Events, 24 Hours Period, $w = 100$, $I = 1$ and $f_{min} = 0.03\%$

In Figure 6 the number of distinct events is increased from 100 to 2000 resulting only in small runtime changes, which are influenced by varying event distributions. Figure 7 shows the runtime if the number of events increases. This condenses the event sequence and increases the number of discovered episodes rapidly, as shown in Figure 7 where the runtime increases to more than 30 minutes.

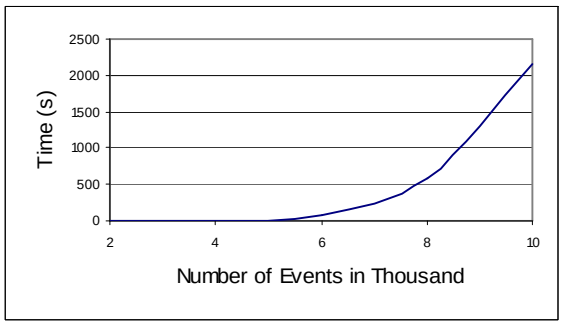


Figure 7: Runtime Characteristics, 300 distinct Events, 24 Hours Period, w = 30, I = 1 and f_{min} = 0.03%

Figure 8 shows the scale-up characteristics for data sets from 10,000 events up to 50,000 events covering a time period of 5 to 25 days (see Table 5). The analysis shows that the runtime increases linear with respect to the size of the input event sequence resulting in execution times between 16 and 58 minutes.

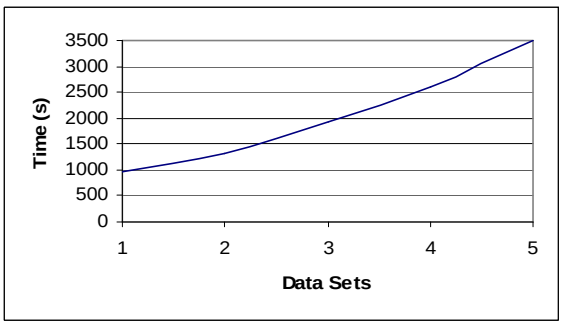


Figure 8: Runtime Characteristics, 300 distinct Events, w = 100, I = 1 and f_{min} = 0.01%

Data Set	Number of Events	Time Period
1	10,000	5 Days
2	20,000	10 Days
3	30,000	15 Days
4	40,000	20 Days
5	50,000	25 Days

Table 5: Data Set Characteristics

7 Conclusions and Further Work

As is the case for all knowledge discovery algorithms the size and structure of the input data set significantly influences the overall runtime. Therefore it is always desirable to keep the number of data base scans to a minimum. Two methods can be used to reduce the number of database scans. Firstly, the current tree is extended by more than one level for each run over the database. Secondly, the second tree storing serial episodes is extended simultaneously. For instance, if a number of parallel episodes are found within a given event window and the tree storing these episodes is fully updated then these episodes can be used directly to growth the tree storing the serial episodes. Both methods would effectively reduce the number of database scans. However, since unwanted patterns can only be removed after a full database scan, the cost is a significant memory overhead of episodes that may not fulfil the specified constraints.

Another method to improve data handling is to mark event windows that do not extent any pattern as inactive and therefore to exclude them from the ongoing discovery process. This also can be used as an *end of discovery* threshold because no more patterns can be found if the number of active event windows is less the support threshold. This also requires a memory overhead, which is however acceptable considering the expected performance improvement.

A novel technique has been presented to effectively discover frequent composite episodes from temporal data. The proposed tree structure provides a flexible and efficient method to store and process episodes, and excludes the requirement for candidate generation.

References

[1] R. Agrawal, S. Srikant: Fast Algorithms for Mining Association Rules, Proc. Of the 20th VLDB Conference, Santiago, Chile, 1994

[2] J.Han, J.Pei; Mining Frequent Patterns by Pattern-Growth: Methodology and Implications; ACM SIGKDD, Dec. 2000.

[3] Cheng Lin, Ching Yun, Ming Chen; Utilizing slice scan and selective hash for episode mining; 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2001)

[4] H. Mannila, H. Toivonen, A. Verkamo; Discovering Frequent Episodes in Sequences; Proceedings of the International Conference on Knowledge Discovery and Data Mining; 1995

[5] H. Mannila, H. Toivonen, A.I. Verkamo. Discovery of Frequent Episodes in Event Sequences, Data Mining and Knowledge Discovery, 1:259 - 289, 1997.