Mining Periodic Patterns with a MDL Criterion

Esther Galbrun, Peggy Cellier, Nikolaj Tatti, Alexandre Termier, Bruno Crémilleux

Mining Periodic Patterns with a MDL Criterion

Esther Galbrun[1], Peggy Cellier[2], Nikolaj Tatti[1,4],
Alexandre Termier[2], and Bruno Crémilleux[3]

[1] Department of Computer Science, Aalto University, Finland
{esther.galbrun,nikolaj.tatti}@aalto.fi
[2] Univ. Rennes, {INSA, Inria}, CNRS, IRISA, France
{peggy.cellier,alexandre.termier}@irisa.fr
[3] Normandie Univ., UNICAEN, ENSICAEN, CNRS – UMR GREYC, France
bruno.cremilleux@unicaen.fr
[4] F-Secure, Finland

Abstract. The quantity of event logs available is increasing rapidly, be they produced by industrial processes, computing systems, or life tracking, for instance. It is thus important to design effective ways to uncover the information they contain. Because event logs often record repetitive phenomena, mining periodic patterns is especially relevant when considering such data. Indeed, capturing such regularities is instrumental in providing condensed representations of the event sequences.
We present an approach for mining periodic patterns from event logs while relying on a Minimum Description Length (MDL) criterion to evaluate candidate patterns. Our goal is to extract a set of patterns that suitably characterises the periodic structure present in the data. We evaluate the interest of our approach on several real-world event log datasets.

Keywords: Periodic patterns · MDL · Sequence mining.

1 Introduction

Event logs are among the most ubiquitous types of data nowadays. They can be machine generated (server logs, database transactions, sensor data) or human generated (ranging from hospital records to life tracking, a.k.a. quantified self), and are bound to become ever more voluminous and diverse with the increasing digitisation of our lives and the advent of the Internet of Things (IoT). Such logs are often the most readily available sources of information on a system or process of interest. It is thus critical to have effective and efficient means to analyse them and extract the information they contain.

Many such logs monitor repetitive processes, and some of this repetitiveness is recorded in the logs. A careful analysis of the logs can thus help understand the characteristics of the underlying recurrent phenomena. However, this is not an easy task: a log usually captures many different types of events. Events related to occurrences of different repetitive phenomena are often mixed together as well as with noise, and the different signals need to be disentangled to allow analysis.

This can be done by a human expert having a good understanding of the domain and of the logging system, but is tedious and time consuming.

Periodic pattern mining algorithms [17] have been proposed to tackle this problem. These algorithms can discover periodic repetitions of sets or sequences of events amidst unrelated events. They exhibit some resistance to noise, when it takes the form of slight variations in the inter-occurrence delay [2] or of the recurrence being limited to only a portion of the data [16]. However, such algorithms suffer from the traditional plague of pattern mining algorithms: they output too many patterns (up to several millions), even when relying on condensed representations [15].

Recent approaches have therefore focused on optimising the quality of the extracted *pattern set* as a whole [5], rather than finding individual high-quality patterns. In this context, the adaptation of the Minimal Description Length (MDL) principle [18, 8] to pattern set mining has given rise to a fruitful line of work [21, 4, 20, 3]. The MDL principle is a concept from information theory based on the insight that any structure in the data can be exploited to compress the data, and aiming to strike a balance between the complexity of the model and its ability to describe the data.

The most important structure of the data on which we focus here, i.e. of event logs, is the periodic recurrence of some events. For a given event sequence, we therefore want to identify a set of patterns that captures the periodic structure present in the data, and we devise a MDL criterion to evaluate candidate pattern sets for this purpose. First, we consider a simple type of model, representing event sequences with cycles over single events. Then, we extend this model so that cycles over distinct events can be combined together. By simply letting our patterns combine not only events but also patterns recursively, we obtain an expressive language of periodic patterns. For instance, it allows us to express the following daily routine:

> Starting Monday at 7:30 AM, wake up, then, 10 minutes later, prepare coffee, repeat every 24 hours for 5 days, repeat this every 7 days for 3 months

as a pattern consisting of two nested cycles, respectively with 24 hours and 7 days periods, over the events "waking up" and "preparing coffee".

In short, we propose a novel approach for mining periodic patterns using a MDL criterion. The main component of this approach—and our main contribution—is the definition of an expressive pattern language and the associated encoding scheme which allows to compute a MDL-based score for a given pattern collection and sequence. We design an algorithm for putting this approach into practise and perform an empirical evaluation on several event log datasets. We show that we are able to extract sets of patterns that compress the input sequences and to identify meaningful patterns.

We start by reviewing the main related work, in Section 2. In Section 3, we introduce our problem setting and a simple model consisting of cycles over single events, which we extend in Section 4. We present an algorithm for mining periodic patterns that compress in Section 5 and evaluate our proposed approach over several event log datasets in Section 6. We reach conclusions in Section 7.

We focus here on the high-level ideas, and refer the interested reader to our report [6] that includes technical details, additional examples and experiments.

2 Related Work

The first approaches for mining periodic patterns used extremely constrained definitions of the periodicity. In [17], *all* occurrences must be regularly spaced; In [10, 9], some missing occurrences are permitted but all occurrences must follow the same regular spacing. As a result, these approaches are extremely sensitive to even small amounts of noise in the data. Ma *et al.* [16] later proposed a more robust approach, which can extract periodic patterns in the presence of gaps of arbitrary size in the data. While the above approaches require time to be discretized as a preprocessing (time steps of hour or day length, for example), several solutions have been proposed to directly discover candidate periods from raw timestamp data, using the Fast Fourier Transform [2] or statistical models [14, 22]. All of the above approaches are susceptible to producing a huge number of patterns, making the exploitation of their results difficult. The use of a *condensed representation* for periodic patterns [15] allows to significantly reduce the number of patterns output, without loss of information, but falls short of satisfactorily addressing the problem.

Considering pattern mining more in general, to tackle this pervasive issue of the overwhelming number of patterns extracted, research has focused on extracting *pattern sets* [5]: finding a (small) set of patterns that together optimise some interest criterion. One such criterion is based on the Minimum Description Length (MDL) principle [7]. Simply put, it states that *the best model is the one that compresses the data best.* Following this principle, the Krimp algorithm [21] was proposed, to select a subset of frequent itemsets that yields the best lossless compression of a transactional database. This algorithm was later improved [19] and the approach extended to analyse event sequences [20, 13, 3]. Along a somewhat different approach, Kiernan and Terzi proposed to use MDL to summarize event sequences [12].

To the best of our knowledge, the only existing method that combines periodic pattern mining and a MDL criterion was proposed by Heierman *et al.* [11]. This approach considers a single regular episode at a time and aims to select the best occurrences for this pattern, independently of other patterns. Instead, we use a MDL criterion in order to select a good collection of periodic patterns.

3 Preliminary Notation and Problem Definition

Next, we formally define the necessary concepts and formulate our problem, focusing on simple cycles.

Event sequences and cycles. Our input data is a collection of timestamped occurrences of some events, which we call an *event sequence*. The events come from an alphabet Ω and will be represented with lower case letters. We assume that an event can occur only once per time step, so the data can be represented as a list of timestamp–event pairs, such as

$$S_1 = \langle (2, c), (3, c), (6, a), (7, a), (7, b), (19, a), (30, a), (31, c), (32, a), (37, b) \rangle \ .$$

Whether timestamps represent days, hours, seconds, or something else depends on the application, the only requirement is that they be expressed as positive integers. We denote as $S^{(\alpha)}$ the event sequence S restricted to event α, that is, the subset obtained by keeping only occurrences of event α.

We denote as $|S|$ the number of timestamp–event pairs contained in event sequence S, i.e. its *length*, and $\Delta(S)$ the time spanned by it, i.e. its *duration*. That is, $\Delta(S) = t_{\text{end}}(S) - t_{\text{start}}(S)$, where $t_{\text{end}}(S)$ and $t_{\text{start}}(S)$ represent the largest and smallest timestamps in S, respectively.

Given such an event sequence, our goal is to extract a representative collection of cycles. A *cycle* is a periodic pattern that takes the form of an ordered list of occurrences of an event, where successive occurrences appear at the same distance from one another. We will not only consider perfect cycles, where the inter-occurrence distance is constant, but will allow some variation.

A cycle is specified by indicating:

- the repeating event, called *cycle event* and denoted as α,
- the number of repetitions of the event, called *cycle length*, r,
- the inter-occurrence distance, called *cycle period*, p, and
- the timestamp of the first occurrence, called *cycle starting point*, τ.

Cycle lengths, cycle periods and cycle starting points take positive integer values (we choose to restrict periods to be integers for simplicity and interpretability). More specifically, we require $r > 1$, $p > 0$ and $\tau \geq 0$.

In addition, since we allow some variation in the actual inter-occurrence distances, we need to indicate an offset for each occurrence in order to be able to reconstruct the original subset of occurrences, that is, to recover the original timestamps. For a cycle of length r, this is represented as an ordered list of $r-1$ signed integer offsets, called the *cycle shift corrections* and denoted as E. Hence, a cycle is a 5-tuple $C = (\alpha, r, p, \tau, E)$.

For a given cycle $C = (\alpha, r, p, \tau, E)$, with $E = \langle e_1, \dots, e_{r-1} \rangle$ we can recover the corresponding occurrences timestamps by reconstructing them recursively, starting from τ: $t_1 = \tau$, $t_k = t_{k-1} + p + e_{k-1}$. Note that this is different from first reconstructing the occurrences while assuming perfect periodicity as $\tau, \tau + p, \tau + 2p, \dots, \tau + (r-1)p$, then applying the corrections, because in the former case the corrections actually accumulate.

Then, we overload the notation and denote the time spanned by the cycle as $\Delta(C)$. Denoting as $\sigma(E)$ the sum of the shift corrections in E, $\sigma(E) = \sum_{e \in E} e$, we have $\Delta(C) = (r-1)p + \sigma(E)$. Note that this assumes that the correction maintains the order of the occurrences. This assumption is reasonable since an alternative cycle that maintains the order can be constructed for any cycle that does not.

We denote as $cover(C)$ the corresponding set of reconstructed timestamp–event pairs $cover(C) = \{(t_1, \alpha), (t_2, \alpha), \dots, (t_r, \alpha)\}$. We say that a cycle covers an occurrence if the corresponding timestamp–event pair belongs to the reconstructed subset $\mathit{cover}(C)$.

Since we represent time in an absolute rather than relative manner and assume that an event can only occur once at any given timestamp, we do not need to worry about overlapping cycles nor about an order between cycles. Given a collection of cycles representing the data, the original list of occurrences can be reconstructed by reconstructing the subset of occurrences associated with each cycle, regardless of order, and taking the union. We overload the notation and denote as $\mathit{cover}(\mathcal{C})$ the set of reconstructed timestamp–event pairs for a collection $\mathcal{C}$ of cycles $\mathcal{C} = \{C_1, \dots, C_m\}$, that is $\mathit{cover}(\mathcal{C}) = \bigcup_{C \in \mathcal{C}} \mathit{cover}(C)$.

For a sequence S and cycle collection $\mathcal{C}$ we call *residual* the timestamp–event pairs not covered by any cycle in the collection: $\mathit{residual}(\mathcal{C}, S) = S \setminus \mathit{cover}(\mathcal{C})$.

We associate a cost to each individual timestamp–event pair $o = (t, \alpha)$ and each cycle C, respectively denoted as $L(o)$ and $L(C)$, which we will define shortly. Then, we can reformulate our problem of extracting a representative collection of cycles as follows:

Problem 1. Given an event sequence S, find the collection of cycles $\mathcal{C}$ minimising the cost

$$L(\mathcal{C}, S) = \sum_{C \in \mathcal{C}} L(C) + \sum_{o \in \mathit{residual}(\mathcal{C}, S)} L(o) \;.$$

Code lengths as costs. This problem definition can be instantiated with different choices of costs. Here, we propose a choice of costs motivated by the MDL principle. Following this principle, we devise a scheme for encoding the input event sequence using cycles and individual timestamp–event pairs. The cost of an element is then the length of the code word assigned to it under this scheme, and the overall objective of our problem becomes finding the collection of cycles that results in the shortest encoding of the input sequence, i.e. finding the cycles that compress the data most. In the rest of this section, we present our custom encoding scheme. Note that all logarithms are to base 2.

For each cycle we need to specify its event, length, period, starting point and shift corrections, that is $L(C) = L(\alpha) + L(r) + L(p) + L(\tau) + L(E)$. It is important to look more closely at the range in which each of these pieces of information takes value, at what values—if any—should be favoured, and at how the values of the different pieces depend on one another.

To encode the cycles' events, we can use codes based on the events' frequency in the original sequence, so that events that occur more frequently in the event sequence will receive shorter code words: $L(\alpha) = -\log(\mathit{fr}(\alpha)) = -\log(\left|S^{(\alpha)}\right| / |S|)$. This requires that we transmit the number of occurrences of each event in the original event sequence. To optimise the overall code length, the length of the code word associated to each event should actually depend on the frequency of the event in the selected collection of cycles. However, this would require keeping track of these frequencies and updating the code lengths dynamically. Instead, we use the frequencies of the events in the input sequence as a simple proxy.

Clearly, a cycle with event α cannot have a length greater than $\left|S^{(\alpha)}\right|$. Once the cycle event α and its number of occurrences are known, we can encode the cycle length with a code word of length $L(r) = \log(\left|S^{(\alpha)}\right|)$, resulting in the same code length for large numbers of repetitions as for small ones.

Clearly, a cycle spans at most the time of the whole sequence, i.e. $\Delta(C) \leq \Delta(S)$, so that knowing the cycle length, the shift corrections, and the sequence time span, we can encode the cycle period with a code word of length

$$L(p) = \log\left(\left\lfloor \frac{\Delta(S) - \sigma(E)}{r-1} \right\rfloor\right) .$$

Next, knowing the cycle length and period as well as the sequence time span, we can specify the value of the starting point with a code word of length

$$L(\tau) = \log(\Delta(S) - \sigma(E) - (r-1)p + 1) .$$

Finally, we encode the shift corrections as follows: each correction e is represented by $|e|$ ones, prefixed by a single bit to indicate the direction of the shift, with each correction separated from the previous one by a zero. For instance, $E = \langle 3, -2, 0, 4 \rangle$ would be encoded as $0\mathit{111}\mathbf{0}1\mathit{11}\mathbf{0}0\mathbf{0}0\mathit{1111}\mathbf{0}$ with value digits, separating digits and sign digits, in italics, bold and normal font, respectively (the sign bit for zero is arbitrarily set to 0 in this case). As a result, the code length for a sequence of shift corrections E is $L(E) = 2\,|E| + \sum_{e \in E} |e|$.

Putting everything together, we can write the cost of a cycle C as

$$\begin{aligned} L(C) = & \log(|S|) + \log\left(\left\lfloor \frac{\Delta(S) - \sigma(E)}{r-1} \right\rfloor\right) \\ & + \log(\Delta(S) - \sigma(E) - (r-1)p + 1) + 2\,|E| + \sum_{e \in E} |e| \ . \end{aligned}$$

On the other hand, the cost of an individual occurrence $o = (t, \alpha)$ is simply the sum of the cost of the corresponding timestamp and event:

$$L(o) = L(t) + L(\alpha) = \log(\Delta(S) + 1) - \log(\left|S^{(\alpha)}\right| / |S|) .$$

Note that if our goal was to actually encode the input sequence, we would need to transmit the smallest and largest timestamps ($t_{\text{start}}(S)$ and $t_{\text{end}}(S)$), the size of the event alphabet ($|\Omega|$), as well as the number of occurrences of each event ($\left|S^{(\alpha)}\right|$ for each event α) of the event sequence. We should also transmit the number of cycles in the collection ($|\mathcal{C}|$), which can be done, for instance with a code word of length $\log(|S|)$. However, since our goal is to compare collections of cycles, we can simply ignore this, as it represents a fixed cost that remains constant for any chosen collection of cycles.

Finally, consider that we are given an ordered list of occurrences $\langle t_1, t_2, \ldots, t_l \rangle$ of event α, and we want to determine the best cycle with which to cover all these occurrences at once. Some of the parameters of the cycle are determined, namely the repeating event α, the length r, and the timestamp of the first occurrence τ. All we need to determine is the period p that yields the shortest code length for the cycle. In particular, we want to find p that minimises $L(E)$.

The shift corrections are such that $E_k = (t_{k+1} - t_k) - p$. If we consider the list of inter-occurrence distances $d_1 = t_2 - t_1, d_2 = t_3 - t_2, \ldots, d_{l-1} = t_l - t_{l-1}$, the problem of finding p that minimises $L(E)$ boils down to minimising $\sum_{d_i} |d_i - p|$. This is achieved by letting p equal the geometric median of the inter-occurrence distances, which, in the one-dimensional case, is simply the median. Hence, for this choice of encoding for the shift corrections, the optimal cycle covering a list of occurrences can be determined by simply computing the inter-occurrences distances and taking their median as the cycle period.

4 Defining Tree Patterns

So far, our pattern language is restricted to cycles over single events. In practise, however, several events might recur regularly together and repetitions might be nested with several levels of periodicity. To handle such cases, we now introduce a more expressive pattern language, that consists of a hierarchy of cyclic blocks, organised as a tree.

Instead of considering simple cycles specified as 5-tuples $C = (\alpha, r, p, \tau, E)$ we consider more general patterns specified as triples $P = (T, \tau, E)$, where T denotes the tree representing the hierarchy of cyclic blocks, while τ and E respectively denote the starting point and shift corrections of the pattern, as with cycles.

Pattern trees. Each *leaf node* in a pattern tree represents a simple block containing one event. Each *intermediate node* represents a cycle in which the children nodes repeat at a fixed time interval. In other words, each intermediate node represents cyclic repetitions of a sequence of blocks. The root of a pattern tree is denoted as B_0. Using list indices, we denote the children of a node B_X as B_{X1}, B_{X2}, etc. All children of an intermediate node except the first one are associated to their distance to the preceding child, called the *inter-block distance*, denoted as d_X for node B_X. Inter-block distances take non-negative integer values. Each intermediate node B_X is associated with the period p_X and length r_X of the corresponding cycle. Each leaf node B_Y is associated with the corresponding occurring event α_Y.

An example of an abstract pattern tree is shown in Fig. 1. We call *height* and *width* of the pattern tree—and by extension of the associated pattern—respectively the number of edges along the longest branch from the root to a leaf node and the number of leaf nodes in the tree.

For a given pattern, we can construct a tree of event occurrences by expanding the pattern tree recursively, that is, by appending to each intermediate node the corresponding number of copies of the associated subtree, recursively. We call this expanded tree the *expansion tree* of the pattern, as opposed to the contracted *pattern tree* that more concisely represents the pattern.

We can enumerate the event occurrences of a pattern by traversing its expansion tree and recording the encountered leaf nodes. We denote as $occs^*(P)$ this list of timestamp–event pairs reconstructed from the tree, prior to correction.

As for the simple cycles, we will not only consider perfect patterns but will allow some variations. For this purpose, a list of shift corrections E is provided with the pattern, which contains a correction for each occurrence except the first one, i.e. $|E| = |occs^*(P)| - 1$. However, as for simple cycles, corrections accumulate over successive occurrences, and we cannot recover the list of corrected occurrences $occs(P)$ by simply adding the individual corrections to the elements of $occs^*(P)$. Instead, we first have to compute the accumulated corrections for each occurrence.

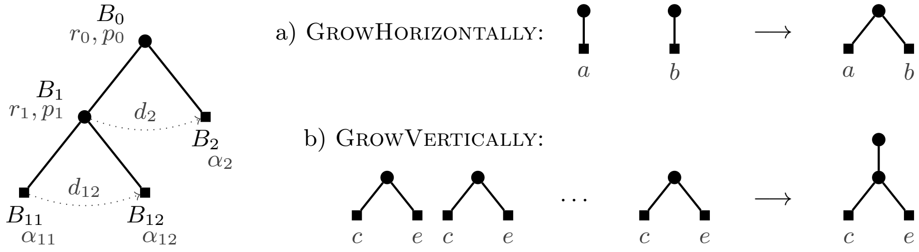

Fig. 1. Abstract pattern tree and examples of growing patterns through combinations.

Encoding the patterns. To transmit a pattern, we need to encode its pattern tree, as well as its starting point and shift corrections. Furthermore, to encode the pattern tree, we consider separately its event sequence, its cycle lengths, its top-level period, and the other values, as explained below.

First we encode the event in the leaves of the pattern tree, traversing the tree from left to right, depth-first. We denote as A the string representing its event sequence. We encode each symbol s in the string A using a code of length $L(s)$, where $L(s)$ depends on the frequency of s, adjusted to take into account the additional symbols '(' and ')', used to delimit blocks. In particular, we set the code lengths for the extended alphabet such that $L(\text{`('}) = L(\text{`)'}) = -\log(1/3)$ for the block delimiters, and $L(\alpha) = -\log(\left|S^{(\alpha)}\right| / (3\,|S|))$ for the original events.

Next, we encode the cycle lengths, i.e. the values r_X associated to each intermediate node B_X encountered while traversing the tree depth-first and from left to right, as a sequence of values, and denote this sequence R. For a block B_X the number of repetitions of the block cannot be larger than the number of occurrences of the least frequent event participating in the block, denoted as $\rho(B_X)$. We can thus encode the sequence of cycle lengths R with code of length

$$L(R) = \sum_{r_X \in R} L(r_X) = \sum_{r_X \in R} \log\left(\rho(B_X)\right) .$$

Knowing the cycle lengths R and the structure of the pattern tree from its event sequence A, we can deduce the total number of events covered by the pattern. The shift corrections for the pattern consist of the correction to each event occurrence except the first one. This ordered list of values can be transmitted using the same encoding as for the simple cycles.

In simple cycles, we had a unique period characterising the distances between occurrences. Instead, with these more complex patterns, we have a period p_X for each intermediate node B_X, as well as an inter-block distance d_X for each node B_X that is not the left-most child of its parent.

First, we transmit the period of the root node of the pattern tree, B_0. In a similar way as with simple cycles, we can deduce the largest possible value for p_0 from r_0 and E. Since we do not know when the events within the main cycle occur, we assume what would lead to the largest possible value for p_0, that is, we assume that all the events within each repetition of the cycle happen at once, so that each repetition spans no time at all.

We denote as D the collection of all the periods (except p_0) and inter-block distances in the tree, that need to be transmitted to fully describe the pattern. To put everything together, the code used to represent a pattern $P = (T, \tau, E)$ has length

$$L(P) = L(A) + L(R) + L(p_0) + L(D) + L(\tau) + L(E) \ .$$

5 Algorithm for Mining Periodic Patterns that Compress

Recall that for a given input sequence S, our goal is to find a collection of patterns $\mathcal{C}$ that minimises the cost

$$L(\mathcal{C}, S) = \sum_{P \in \mathcal{C}} L(P) + \sum_{o \in \mathit{residual}(\mathcal{C}, S)} L(o) \ .$$

It is useful to compare the cost of different patterns, or sets of patterns, on a subset of the data, i.e. compare $L(\mathcal{C}', S')$ for different sets of patterns $\mathcal{C}'$ and some subsequence $S' \subseteq S$. In particular, we might compare the cost of a pattern P to the cost of representing the same occurrences separately. This means comparing

$$L(\{P\}, \mathit{cover}(P)) = L(P) \quad \text{and} \quad L(\emptyset, \mathit{cover}(P)) = \sum_{o \in \mathit{cover}(P)} L(o) \ .$$

If $L(\{P\}, \mathit{cover}(P)) < L(\emptyset, \mathit{cover}(P))$, we say that pattern P is *cost-effective*. In addition, we compare patterns in terms of their cost-per-occurrence ratio defined, for a pattern P, as $L(P)/\left|\mathit{cover}(P)\right|$, and say that a pattern is more *efficient* when this ratio is smaller.

A natural way to build patterns is to start with the simplest patterns, i.e. cycles over single events, and combine them together into more complex, possibly multi-level multi-event patterns.

Assume that we have a pattern tree T_I which occurs multiple times in the event sequence. In particular, assume that it occurs at starting points τ_1, τ_2, …, τ_{r_J} and that this sequence of starting points itself can be represented as a cycle of length r_J and period p_J. In such a case, the occurrences of T_I might be combined together and represented as a nested pattern tree. GrowVertically is the procedure which takes as input a collection $\mathcal{C}_I$ of patterns over a tree T_I

Algorithm 1 Mining periodic patterns that compress.

```
Require: A multi-event sequence $S$, a number $k$ of top candidates to keep
Ensure: A collection of patterns $\mathcal{P}$
1: $\mathcal{I} \leftarrow$ ExtractCycles($S, k$)
2: $\mathcal{C} \leftarrow \emptyset; \mathcal{V} \leftarrow \mathcal{I}; \mathcal{H} \leftarrow \mathcal{I}$
3: while $\mathcal{H} \neq \emptyset$ or $\mathcal{V} \neq \emptyset$ do
4:     $\mathcal{V}' \leftarrow$ CombineVertically($\mathcal{H}, \mathcal{P}, S, k$)
5:     $\mathcal{H}' \leftarrow$ CombineHorizontally($\mathcal{V}, \mathcal{P}, S, k$)
6:     $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{H} \cup \mathcal{V}; \mathcal{V} \leftarrow \mathcal{V}'; \mathcal{H} \leftarrow \mathcal{H}'$
7: $\mathcal{P} \leftarrow$ GreedyCover($\mathcal{C}, S$)
8: return $\mathcal{P}$
```

and returns the nested pattern obtained by combining them together as depicted in Fig. 1(b).

On the other hand, given a collection of patterns that occur close to one another and share similar periods, we might want to combine them together into a concatenated pattern by merging the roots of their respective trees. GrowHorizontally is the procedure which takes as input a collection of patterns and returns the pattern obtained by concatenating them together in order of increasing starting points as depicted in Fig. 1(a).

As outlined in Algorithm 1, our proposed algorithm consists of three stages: *(i)* extracting cycles (line 1), *(ii)* building tree patterns from cycles (lines 2–6) and *(iii)* selecting the final pattern collection (line 7). We now present each stage in turn at a high-level.

Extracting cycles. Considering each event in turn, we use two different routines to mine cycles from the sequence of timestamps obtained by restricting the input sequence to the event of interest, combine and filter their outputs to generate the set of initial candidate patterns. The first routine, ExtractCyclesDP, uses dynamic programming. Indeed, if we allow neither gaps in the cycles nor overlaps between them, finding the best set of cycles for a given sequence corresponds to finding an optimal segmentation of the sequence, and since our cost is additive over individual cycles, we can use dynamic programming to solve it optimally [1]. The second routine, ExtractCyclesTri, extracts cycles using a heuristic which allows for gaps and overlappings. It collects triples (t_0, t_1, t_2) such that $||t_2 - t_1| - |t_1 - t_0|| \leq \ell$, where ℓ is set so that the triple can be beneficial when used to construct longer cycles. Triples are then chained into longer cycles. Finally, the set $\mathcal{C}$ of cost-effective cycles obtained by merging the output of the two routines is filtered to keep only the k most efficient patterns for each occurrence for a user-specified k, and returned.

Building tree patterns from cycles. The second stage of the algorithm builds tree patterns, starting from the cycles produced in the previous stage. That is, while there are new candidate patterns, the algorithm performs combination rounds, trying to generate more complex patterns through vertical and horizontal combinations. If desired, this stage can be skipped, thereby restricting the pattern language to simple cycles.

In a round of vertical combinations performed by CombineVertically (line 4), each distinct pattern tree represented among the new candidates in $\mathcal{H}$ is considered in turn. Patterns over that tree are collected and ExtractCyclesTri is used to mine cycles from the corresponding sequence of starting points. For each obtained cycle, a nested pattern is produced by combining the corresponding candidates using GrowVertically (see Fig. 1(b)).

In a round of horizontal combinations performed by CombineHorizontally (line 5), a graph G is constructed, with vertices representing candidate patterns and with edges connecting pairs of candidates $\mathcal{K} = \{P_I, P_J\}$ for which the concatenated pattern $P_N = \text{GrowHorizontally}(\mathcal{K})$ satisfies $L(\{P_N\}, \mathit{cover}(\mathcal{K})) < L(\mathcal{K}, \mathit{cover}(\mathcal{K}))$. A new pattern is then produced for each clique of G, by applying GrowHorizontally to the corresponding set of candidate patterns.

Selecting the final pattern collection. Selecting the final set of patterns to output among the candidates in $\mathcal{C}$ is very similar to solving a weighted set cover problem. Therefore, the selection is done using a simple variant of the greedy algorithm for this problem, denoted as GreedyCover (line 7).

6 Experiments

In this section, we evaluate the ability of our algorithm to find patterns that compress the input event sequences. We make the code and the prepared datasets publicly available.[5] To the best of our knowledge, no existing algorithm carries out an equivalent task and we are therefore unable to perform a comparative evaluation against competitors. To better understand the behaviour of our algorithm, we first performed experiments on synthetic sequences. We then applied our algorithm to real-world sequences including process execution traces, smartphone applications activity, and life-tracking. We evaluate our algorithm's ability to compress the input sequences and present some examples of extracted patterns.

For a given event sequence, the main objective of our algorithm is to mine and select a good collection of periodic patterns, in the sense that the collection should allow to compress the input sequence as much as possible. Therefore, the main measure that we consider in our experiments is the *compression ratio*, defined as the ratio between the length of the code representing the input sequence with the considered collection of patterns and the length of the code representing the input sequence with an empty collection of patterns, i.e. using only individual event occurrences, given as a percentage. For a given sequence S and collection of patterns $\mathcal{C}$ the compression ratio is defined as

$$\%L = 100 \cdot L(\mathcal{C}, S)/L(\emptyset, S) \ ,$$

with smaller values associated to better pattern collections.

[5] `https://github.com/nurblageij/periodic-patterns-mdl`

Table 1. Statistics of the event log sequences used in the experiments.

	$\|S\|$	$\Delta(S)$	$\|\Omega\|$	$\left\|S^{(\alpha)}\right\|$		$L(\emptyset, S)$	RT (s)	
				med	max		cycles	overall
`3zap`	181644	181643	443	22	36697	4154277	2094	35048
`bugzilla`	16775	16774	91	6	3332	303352	112	522
`samba`	28751	7461	119	44	2905	520443	214	2787
`sacha`	65977	221445	141	231	4389	1573140	2963	14377
`ubiqLog` (31 sequences)								
min	413	11391	10	23	194	6599	1	1
median	23859	87591	87	52	2131	486633	232	1020
max	167863	17900307	241	129	6101	3733349	2297	28973

Datasets. Our first two datasets come from a collaboration with STMicroelectronics and are execution traces of a set-top box based on the STiH418 SoC[6] running STLinux. Both traces are a log of system actions (interruptions, context switches and system calls) taken by the KPTrace instrumentation system developed at STMicroelectronics. The `3zap` sequence corresponds to 3 successive changes of channel ("zap"), while the `bugzilla` sequence corresponds to logging a display blackout bug into the bug tracking system of ST. For our analysis of these traces, we do not consider timestamps, only the succession of events.

The `ubiqLog` dataset was obtained from the UCI Machine learning repository.[7] It contains traces collected from the smartphones of users over the course of two months. For each of 31 users we obtain a sequence recording what applications are run on that user's smartphone. We consider absolute timestamps with a granularity of one minute.

The `samba` dataset consists of a single sequence recording the emails identifying the authors of commits on the git repository of the samba network file system[8] from 1996 to 2016. We consider timestamps with a granularity of one day. We aggregated together users that appeared fewer than 10 times.

The `sacha` dataset consists of a single sequence containing records from the *quantified awesome* life log[9] recording the daily activities of its author between November 2011 and January 2017. The daily activities are associated to start and end timestamps, and are divided between categories organised into a hierarchy. Categories with fewer than 200 occurrences were aggregated to their parent category. Each resulting category is represented by an event. Adjacent occurrences of the same event were merged together. We consider absolute timestamps with a granularity of 15 minutes.

[6] STiH418 description: `http://www.st.com/resource/en/data_brief/stih314.pdf`

[7] `https://archive.ics.uci.edu/ml/datasets/UbiqLog+(smartphone+lifelogging)`

[8] `https://git.samba.org/`

[9] `http://quantifiedawesome.com/records`

Table 2. Summary of results for the separate event sequences.

	$\%L$	$L{:}\mathcal{R}$	s	v	h	m	c^+		$\%L$	$L{:}\mathcal{R}$	s	v	h	m	c^+
			`3zap`								`bugzilla`				
$\mathcal{C}_S$	56.32	0.41	11852	–	–	–	2325	$\mathcal{C}_S$	48.58	0.12	262	–	–	–	1652
$\mathcal{C}_V$	55.14	0.40	10581	581	–	–	2325	$\mathcal{C}_V$	48.56	0.12	259	1	–	–	1652
$\mathcal{C}_H$	47.84	0.35	3459	–	4912	–	2325	$\mathcal{C}_H$	42.43	0.12	133	–	70	–	1652
$\mathcal{C}_{V+H}$	47.40	0.34	3499	419	4302	–	2325	$\mathcal{C}_{V+H}$	42.39	0.12	130	1	72	–	1652
$\mathcal{C}_F$	46.99	0.34	3499	91	4154	268	2325	$\mathcal{C}_F$	42.41	0.13	124	1	70	2	1652
			`samba`								`sacha`				
$\mathcal{C}_S$	28.42	0.14	429	–	–	–	2657	$\mathcal{C}_S$	74.34	0.37	9602	–	–	–	304
$\mathcal{C}_F$	28.37	0.13	409	0	17	0	2657	$\mathcal{C}_F$	68.64	0.35	3957	0	2996	0	582

Fig. 2. Compression ratios for the sequences from the `ubiqLog` dataset.

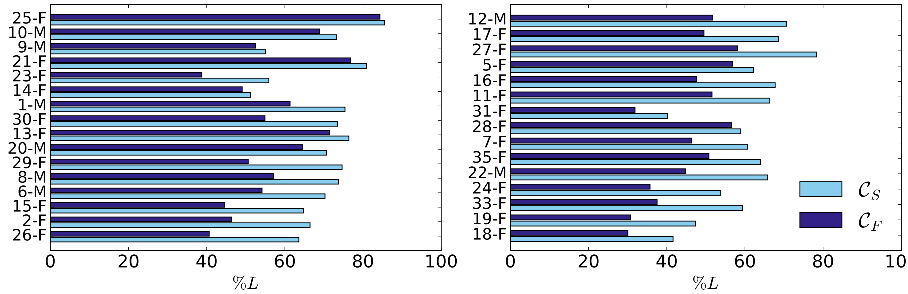

Table 1 presents the statistics of the sequences used in our experiments.

We indicate the length ($|S|$) and duration ($\Delta(S)$) of each sequence, the size of its alphabet ($|\Omega|$), as well as the median and maximum length of the event subsequences ($\left|S^{(\alpha)}\right|$). We also indicate the code length of the sequence when encoded with an empty collection of patterns ($L(\emptyset, S)$), as well as the running time of the algorithm (RT, in seconds) for mining and selecting the patterns, as well as for the first stage of mining cycles for each separate event.

Measures. Beside the compression ratio ($\%L$) achieved with the selected pattern collections, we consider several other characteristics (see Table 2). For a given pattern collection $\mathcal{C}$, we denote the set of residuals $\mathit{residual}(\mathcal{C}, S)$ simply as $\mathcal{R}$ and look at what fraction of the code length is spent on them, denoted as $L : \mathcal{R} = \sum_{o \in \mathcal{R}} L(o)/L(\mathcal{C}, S)$. We also look at the number of patterns of different types in $\mathcal{C}$: (s) simple cycles, i.e. patterns with width $= 1$ and height $= 1$, (v) vertical patterns, with width $= 1$ and height > 1, (h) horizontal patterns, with width > 1 and height $= 1$, and (m) proper two-dimensional patterns, with width > 1 and height > 1. Finally, we look at the maximum cover size of patterns in $\mathcal{C}$, denoted as c^+.

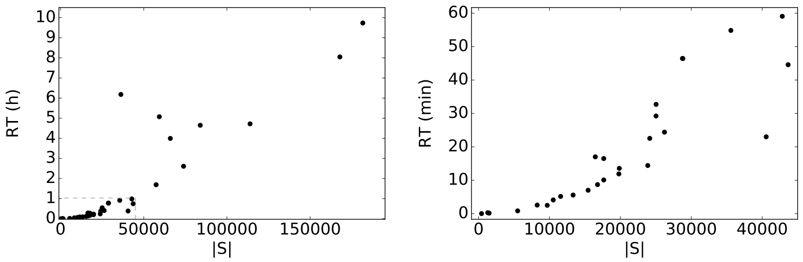

Fig. 3. Running times for mining the different sequences (in hours, left) and zooming in on shorter sequences (in minutes, right).

Results. In addition to the final collection of patterns returned by the algorithm after potentially a few rounds of combinations (denoted as $\mathcal{C}_F$), we also consider intermediate collections of patterns, namely a collection selected among cycles mined during the first stage of the algorithm (denoted as $\mathcal{C}_S$), including patterns from the first round of horizontal combinations ($\mathcal{C}_H$), of vertical combinations ($\mathcal{C}_V$), and both, i.e. at the end of the first round of combinations ($\mathcal{C}_{V+H}$).

A summary of the results for the separate event sequences `3zap`, `bugzilla`, `samba` and `sacha`, is presented in Table 2. Fig. 2 shows the compression ratios achieved on event sequences from the `ubiqLog` dataset.

We see that the algorithm is able to find sets of patterns that compress the input event sequences. The compression ratio varies widely depending on the considered sequence, from a modest 84% for some sequences from `ubiqLog` to a reduction of more than two thirds, for instance for `samba`. To an extent, the achieved compression can be interpreted as an indicator of how much periodic structure is present in the sequence (at least of the type that can be exploited by our proposed encoding and detected by our algorithm). In some cases, as with `samba`, the compression is achieved almost exclusively with simple cycles, but in many cases the final selection contains a large fraction of horizontal patterns (sometimes even about two thirds), which bring a noticeable improvement in the compression ratio (as can be seen in Fig. 2, for instance). Vertical patterns, on the other hand, are much more rare, and proper two-dimensional patterns are almost completely absent. The `bugzilla` sequence features such patterns, and even more so the `3zap` sequence. This agrees with the intuition that recursive periodic structure is more likely to be found in execution logs tracing multiple recurrent automated processes.

Fig. 3 shows the running times for sequences from the different datasets. The running times vary greatly, from only a few seconds to several hours. Naturally, mining longer sequences tends to require longer running times.

Example patterns. Finally, we present some examples of patterns obtained from the `sacha` and `3zap` sequences, in Fig. 4. The start and end of an activity A are denoted as "[A" and "A]" respectively. The patterns from the `sacha` sequence are simple and rather obvious, but they make sense when considering

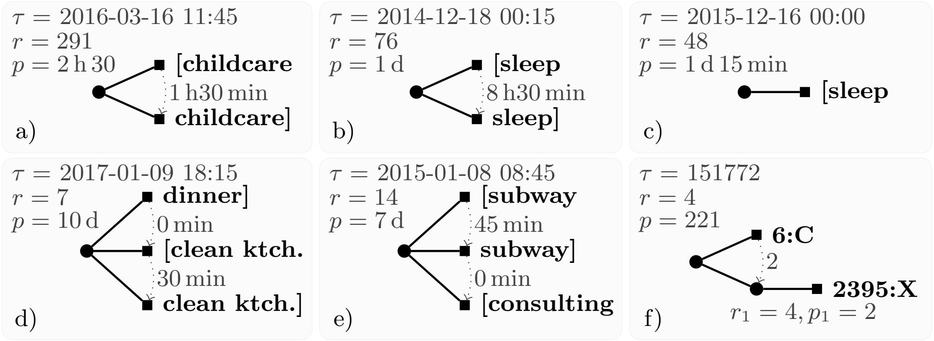

Fig. 4. Example patterns from `sacha` (a–e) and `3zap` (f).

everyday activities. The fact that we are able to find them is a clear sign that the method is working. The `3zap` pattern is a typical system case: the repetition of a context switch (6:C) followed by several activations of a process (2395:X).

Most of the discovered patterns are fairly simple. We suspect that this is due to the nature of the data: there are no significantly complex patterns in these event log sequences. In any case, the expressivity of our proposed pattern language comes at no detriment to the simpler, more common patterns, but brings the potential benefit of identifying sequences containing exceptionally regular structure.

7 Conclusion

In this paper, we propose a novel approach for mining periodic patterns with a MDL criterion, and an algorithm to put it into practise. Through our experimental evaluation, we show that we are able to extract sets of patterns that compress the input event sequences and to identify meaningful patterns.

How to take prior knowledge into account is an interesting question to explore. Making the algorithm more robust to noise and making it more scalable using for instance parallelisation, are some pragmatic directions for future work, as is adding a visualisation tool to support the analysis and interpretation of the extracted patterns in the context of the event log sequence.

Acknowledgements

The authors thank Hiroki Arimura and Jilles Vreeken for valuable discussions. This work has been supported by Grenoble Alpes Metropole through the Nano2017 Itrami project, by the QCM-BioChem project (CNRS Mastodons) and by the Academy of Finland projects “Nestor” (286211) and “Agra” (313927).

References

1. R. Bellman. On the approximation of curves by line segments using dynamic programming. *Communications of the ACM*, 4(6), 1961.
2. C. Berberidis, I. P. Vlahavas, W. G. Aref, M. J. Atallah, and A. K. Elmagarmid. On the discovery of weak periodicities in large time series. In *PKDD'02*, pages 51–61, 2002.
3. A. Bhattacharyya and J. Vreeken. Efficiently summarising event sequences with rich interleaving patterns. In *SDM'17*, pages 795–803. SIAM, 2017.
4. F. Bonchi, M. van Leeuwen, and A. Ukkonen. Characterizing uncertain data using compression. In *SDM'11*, pages 534–545. SIAM, 2011.
5. L. De Raedt and A. Zimmermann. Constraint-based pattern set mining. In *SDM'07*, pages 237–248. SIAM, 2007.
6. E. Galbrun, P. Cellier, N. Tatti, A. Termier, and B. Crémilleux. Mining periodic patterns with a MDL criterion. *ArXiv e-prints*, 2018. arXiv:1807.01706 [cs.DB].
7. P. Grünwald. Model selection based on minimum description length. *Journal of Mathematical Psychology*, 44(1):133–152, 2000.
8. P. Grünwald. *The Minimum Description Length Principle*. MIT Press, 2007.
9. J. Han, G. Dong, and Y. Yin. Efficient mining of partial periodic patterns in time series database. In *ICDE'99*, pages 106–115, 1999.
10. J. Han, W. Gong, and Y. Yin. Mining segment-wise periodic patterns in time-related databases. In *KDD'98*, pages 214–218, 1998.
11. E. O. Heierman, III and D. J. Cook. Improving home automation by discovering regularly occurring device usage patterns. In *ICDM'03*, pages 537–540, 2003.
12. J. Kiernan and E. Terzi. Constructing comprehensive summaries of large event sequences. *ACM Trans. Knowl. Discov. Data*, 3(4):21:1–21:31, 2009.
13. H. T. Lam, F. Moerchen, D. Fradkin, and T. Calders. Mining compressing sequential patterns. In *SDM'12*, pages 319–330. SIAM, 2012.
14. Z. Li, J. Wang, and J. Han. Mining event periodicity from incomplete observations. In *KDD'12*, pages 444–452. ACM, 2012.
15. P. Lopez-Cueva, A. Bertaux, A. Termier, J.-F. Méhaut, and M. Santana. Debugging embedded multimedia application traces through periodic pattern mining. In *Int. Conf. on Embedded Software, EMSOFT'12*, 2012.
16. S. Ma and J. L. Hellerstein. Mining partially periodic event patterns with unknown periods. In *ICDE'01*, pages 205–214. IEEE Computer Society, 2001.
17. B. Özden, S. Ramaswamy, and A. Silberschatz. Cyclic association rules. In *ICDE'98*, pages 412–421. IEEE Computer Society, 1998.
18. J. Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, 1978.
19. K. Smets and J. Vreeken. Slim: Directly mining descriptive patterns. In *SDM'12*, pages 236–247. SIAM, 2012.
20. N. Tatti and J. Vreeken. The long and the short of it: Summarising event sequences with serial episodes. In *KDD'12*, pages 462–470. ACM, 2012.
21. J. Vreeken, M. van Leeuwen, and A. Siebes. Krimp: Mining itemsets that compress. *Data Min Knowl Discov*, 23(1):169–214, 2011.
22. Q. Yuan, W. Zhang, C. Zhang, X. Geng, G. Cong, and J. Han. Pred: Periodic region detection for mobility modeling of social media users. In *WSDM'17*, pages 263–272. ACM, 2017.