EpisodeSupport: a Global Constraint for Mining Frequent Patterns in a Long Sequence of Events

Quentin Cappart, John O. R. Aoga*, and Pierre Schaus

Université catholique de Louvain, Louvain-La-Neuve, Belgium
{quentin.cappart|john.aoga|pierre.schaus}@uclouvain.be

Abstract. The number of applications generating sequential data is exploding. This work studies the discovering of frequent patterns in a large sequence of events, possibly time-stamped. This problem is known as the Frequent Episode Mining (FEM). Similarly to the mining problems recently tackled by Constraint Programming (CP), FEM would also benefit from the modularity offered by CP to accommodate easily additional constraints on the patterns. These advantages do not offer a guarantee of efficiency. Therefore, we introduce two global constraints for solving FEM problems with or without time consideration. The time-stamped version can accommodate gap and span constraints on the matched sequences. Our experiments on real data sets of different levels of complexity show that the introduced constraints is competitive with the state-of-the-art methods in terms of execution time and memory consumption while offering the flexibility of adding constraints on the patterns.

1 Introduction

The trend in data science is to automate the data-analysis as much as possible. Examples are the *Automating machine learning* project [10], or the commercial products `www.automaticstatistician.com` and `www.datarobot.com`. The Auto-Weka [18] and Auto-sklearn [9] modules can automate the selection of a machine learning algorithm and its parameters for solving standard classification or regression tasks. Most of these automated tools target tabular datasets, but not yet sequences and time-series data. Data-mining problems on sequences and time series remain challenging [32] but are nevertheless of particular interest [7, 29]. We believe that Constraint Programming (CP), because of the flexibility it offers, may play a role in the portfolio of techniques available for automating data-science on sequential data. As an illustration of this flexibility, Negrevergne and Guns [23] identified some constraints that could be stated on the patterns to discover in a database of sequences: length, exclusion/inclusion on symbols, membership to a regular language [25], etc. The idea of using CP for data-mining is not new. It was already used for item-set mining [11, 12, 24, 28], for Sequential Pattern Mining (SPM) [2, 3, 16, 23] or even for mobility profile mining [17].

In this paper we address the Frequent Episode Mining (FEM), first introduced with the apriori-like method WINEPI [22] and improved on MINEPI [21],

* This author is supported by the FRIA-FNRS, Belgium.

with a CP approach. Contrarily to the traditional SPM, FEM aims at discovering frequent patterns in a single but very long sequence of symbols possibly time-stamped. Assume for instance a non time-stamped sequence $\langle a, b, a, c, b, a, c \rangle$ and we are looking for patterns of length three occurring at least two times. Such a subsequence is $\langle a, b, c \rangle$ that occurs exactly two times. A first occurrence is $\langle \mathbf{a}, \mathbf{b}, a, \mathbf{c}, b, a, c \rangle$ and a second one is $\langle a, b, \mathbf{a}, c, \mathbf{b}, a, \mathbf{c} \rangle$. The attentive reader may wonder why $\langle \mathbf{a}, \mathbf{b}, a, c, b, a, \mathbf{c} \rangle$ is not counted. The reason is that the *head/total frequency* measure [15] avoids duplicate counting by restricting a counting position to the first one. This measure has some interesting properties such as the well known anti-monotonicity which states that if a sequence is frequent all its subsequences are frequent too and reversely. This property makes it possible to design faster data-mining algorithm. Indeed, based on these properties, Huang and Chang [14] proposed two algorithms, MINEPI+ and EMMA. While the first one is only a small adaptation of MINEPI [21], the second uses memory anchors in order to accelerate the mining task with the price of a greater memory consumption. As variants of this problem, episodes can be closed [30, 34], and other (interestingness) measures [4, 6, 19] can be considered. When considering time-stamped sequences such as $\langle (a, 1), (b, 3), (a, 5), (c, 6), (b, 7), (a, 8), (c, 14) \rangle$, one may also want to impose time constraints on the time difference between any two matched symbols or between the first and last matched ones. Such constraints, called *gap* and *span* were also introduced for the SPM [3] with CP.

The problem of discovering frequent pattern in a very long sequence can be reduced do the standard SPM problem [1]. The reduction consists in creating a database of sequences composed of all the suffixes of the long sequence. For our example, the sequence database would be: $\langle a, b, a, c, b, a, c \rangle$, $\langle b, a, c, b, a, c \rangle$, $\langle a, c, b, a, c \rangle$, $\langle c, b, a, c \rangle$, $\langle b, a, c \rangle$, $\langle a, c \rangle$, $\langle c \rangle$. A small adaptation of existing algorithms is required though to match any sequence of the database on its first position in accordance with the *head/total frequency* measure. This reduction has one main drawback. The spatial complexity is $\mathcal{O}(n^2)$ with n the length of the sequence. Such a complexity will quickly exceed the available memory for sequence lengths as small as a few thousands.

The contribution of this paper is a flexible and efficient approach for solving the frequent episode mining problem. WINEPI, MINEPI and EMMA are specialized algorithms not able to accommodate additional constraints. We introduce two global constraints for FEM, which use an implicit decomposition having a $\mathcal{O}(n)$ spatial complexity. Our global constraints are inspired by the state-of-the-art approaches [2, 3, 16] but keeping the reduction into a suffix database implicit instead of explicit. We propose two versions: with and without considering *gap* and *span* constraints. We are also able to take some algorithmic advantages in the filtering algorithms using the property that the (implicit) database is composed of sorted suffixes from a same sequence. To the best of our knowledge, this work is the first CP-based approach proposed for solving efficiently this family of problems with the benefit that several other constraints can be added.

This paper is organized as follows. Section 2 introduces the technical background related to the FEM problem. It explains how the problem can be modeled using CP and presents our first global constraint (`episodeSupport`). Section 3 shows how time can be integrated into the problem and describes the second global constraint (`episodeSupport` with time). Finally, experiments are carried out on synthetic and real-life datasets in Sect. 4.

2 Mining Episodes in a Non Timed Sequence

2.1 Technical Background

Let $\Sigma = \{1, \ldots, L\}$ be an alphabet representing a set of possible symbols. We define a non timed sequence $s = \big\langle s_1, \ldots, s_n \big\rangle$ over Σ as an ordered list of symbols such that $\forall i \in [1, n], s_i \in \Sigma$. Let us consider the following definitions based on the formalization of Aoga et al. [2] and Huang and Chang [14].

Definition 1 (Subsequence relation, Embedding). *$\alpha = \langle \alpha_1, \ldots, \alpha_m \rangle$ is a subsequence of $s = \langle s_1, \ldots, s_n \rangle$, denoted by $a \preceq s$, if $m \leq n$ and if there exists a list of indexes $(e_1, \ldots, e_m)$ with $1 \leq e_1 \leq \cdots \leq e_m \leq n$ such that $s_{e_i} = \alpha_i$. Such a list is referred as an embedding of s. Sequence s is also referred as a super-sequence of α.*

Example 1. $\langle a, b, c \rangle$ is a subsequence of the sequence $s = \langle a, b, a, c, b, a, c \rangle$ with embeddings $(1, 2, 4)$ or $(1, 2, 7)$ or $(3, 5, 7)$.

Definition 2 (Episode-embedding). *Let us consider $\alpha = \langle \alpha_1, \ldots, \alpha_m \rangle \preceq s$. Embedding $(e_1, \ldots, e_m)$ is an episode-embedding if it is an embedding of s and if all the other embeddings $(e_1, e'_2, \ldots, e'_m)$ are such that $(e_2, \ldots, e_m) \preceq_L (e'_2, \ldots, e'_m)$ where $\preceq_L$ represents a lexicographic ordering.*

Example 2. $\langle a, b, c \rangle$ is a subsequence of s with $(1, 2, 4)$ and $(3, 5, 7)$ as episode-embeddings. Besides, $(1, 2, 7)$ is not an episode-embedding because $(2, 7)$ is lexicographically greater than $(2, 4)$.

Definition 3 (Support). *The support $\sigma_s(\alpha)$ of a subsequence α in a sequence s is the number of episode-embeddings of α in s.*

Example 3. For s of Example 1, we have $\sigma_s(\langle a, b, c \rangle) = 2$.

Frequent Episode Mining (FEM) problem can then be formalised. Let us underline that this definition is related to the *total frequency* measure introduced by Iwanuma et al. [15]. The goal is to count up occurrences without duplication. To do so, we use the concept of prefix-projection introduced in PrefixSpan [13] and used thereafter by Kemmar et al. [16] and Aoga et al. [2] for SPM.

Definition 4 (Frequent Episode Mining (FEM)). *Given a set of symbols Σ, a sequence s over Σ and a threshold θ, the goal is to find all the subsequences α in s such that $\sigma_s(\alpha) \geq \theta$. These subsequences are called episodes.*

Definition 5 (Prefix, Projection, Suffix). *Let $\alpha = \langle \alpha_1, \dots, \alpha_k \rangle$ and $s = \langle s_1, \dots, s_n \rangle$ be two sequences. If $\alpha \preceq s$, then the prefix of s w.r.t. α is the smallest prefix of s that remains a super-sequence of α. Formally, it is the sequence $\langle s_1, \dots, s_j \rangle$ such that $\alpha \preceq \langle s_1, \dots, s_j \rangle$ and such that there exists no $j' < j$ where $\alpha \preceq \langle s_1, \dots, s_{j'} \rangle$. The sequence $\langle s_{j+1}, \dots, s_n \rangle$ is then called the suffix of s w.r.t. α, or the α-projection, and is denoted by $s|_\alpha$. If α is not a subsequence of s, the α-projection is empty.*

Example 4. Given sequence s of Example 1 and $\alpha = \langle b \rangle$, sequence $\langle a, b \rangle$ is a prefix of s w.r.t. α and $\langle a, c, b, a, c \rangle$ is a suffix ($s|_\alpha = \langle a, c, b, a, c \rangle$).

Definition 6 (Initial Projection). *An initial projection of a sequence $s = \langle s_1, \dots, s_n \rangle$ w.r.t. a symbol x, denoted by $s|_x^{\mathcal{I}}$, is the list of all the suffixes $s' = \langle s_i, \dots, s_n \rangle$ such that $s_{i-1} = x$ for all $i \in (1, n]$.*

Example 5. For s and a symbol a, we have $s|_a^{\mathcal{I}} = \big[\langle b, a, c, b, a, c \rangle, \langle c, b, a, c \rangle, \langle c \rangle\big]$.

Definition 7 (Internal Projection). *Given a list of sequences Ω, an internal projection of Ω w.r.t. pattern α, denoted by $\Omega|_\alpha$, is the list of the α-projection of all sequences in Ω. All the empty sequences are removed from $\Omega|_\alpha$.*

Example 6. For $\alpha = \langle b \rangle$ and $\Omega = \big[\langle b, a, c, b, a, c \rangle, \langle c, b, a, c \rangle, \langle c \rangle\big]$, we obtain $\Omega|_\alpha = \big[\langle a, c, b, a, c \rangle, \langle a, c \rangle\big]$.

Definition 8 (Projected Frequency). *Given the list of sequences Ω, and a projection $s|_\alpha$ for each sequence $s \in \Omega$, the projected frequency of a symbol is the number of α-projected sequences where the symbol appears.*

Example 7. Given the internal projection $\Omega|_\alpha$ of Example 6, the projected frequencies are $a : 2, b : 1$ and $c : 2$.

In practice, the initial projections and internal projections can be efficiently stored as a list of pointers in the original sequence s. In our example ($s = \langle a, b, a, c, b, a, c \rangle$), we have $s|_a^{\mathcal{I}} = [2, 4, 7]$ and starting from $\Omega = s|_a^{\mathcal{I}}$ we can represent $\Omega|_{\langle b \rangle} = \big[3, 6\big]$. This representation introduced in PrefixSpan [13] is called the *pseudo projection* representation. The algorithm works as follows. It starts from the empty pattern and successively extends it in a *depth-first search*. At each step, a symbol is added to the pattern, and all the sequences of the database are projected accordingly. A backtrack occurs when all the *projected frequencies* are below the support threshold. When a backtracking is performed during the search, the last appended symbol is removed. This procedure is known as the *pattern growth method* [13]. A new projection is thus built and stored at each step. An important consideration for the efficiency of this method is that the projected sequences do not need to be computed from scratch at each iteration. Instead, the pseudo-projection representation is used and maintained incrementally at each symbol extension of the pattern. Starting from the previous pseudo-projection, when the next symbol is appended, one can start from each position in the pseudo-projection representation and look, for each one, the next matching positions in s equal to this symbol. The new matching positions constitute the new pseudo-projection representation. Since the search follows a depth-first-search strategy, the pseudo projections can be stacked on a same vector allowing to reuse allocated entries on backtrack. This memory management is known as a trailing in CP and was introduced for SPM by Aoga et al. [2, 3].

2.2 Problem Modelling

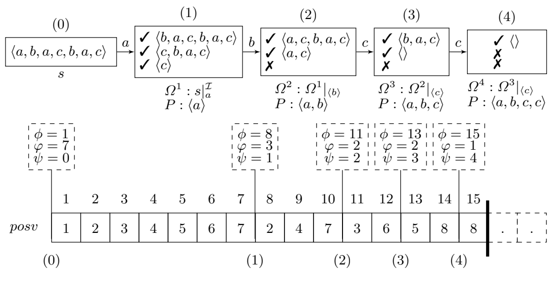
Our first contribution is a global constraint, `episodeSupport`, dedicated to find frequent patterns (or *episodes* [22]) in a sequence without considering time. Let $s = \langle s_1, \ldots, s_n \rangle$ be a sequence of n symbols over Σ, the set of distinct symbols appearing in s, and θ, the minimum support threshold desired.

Decision Variables Let $P = \langle P_1, \ldots, P_n \rangle$ be a sequence of variables representing a pattern. The domain of each variable is defined as $P_i = \Sigma \cup \{\varepsilon\}$ for all $i \in [1, n]$. It indicates that each variable can take any symbol appearing in s as value in addition to ε, which is defined as the empty symbol. An assignment of P_i to ε means that P_i has matched no symbol. It is used to model patterns having a length lower than n. A solution is an assignation of each variable in P.

EpisodeSupport Constraint The `episodeSupport`$(P,\ s,\ \theta)$ constraint enforces the three following constraints: (1) $P_1 \neq \varepsilon$, (2) $P_i = \varepsilon \rightarrow P_{i+1} = \varepsilon$, $\forall i \in \big[2, n\big)$ and (3) $\sigma_s(P) \geq \theta$. The first constraint states that a pattern cannot begin with the empty symbol. It indicates that a valid pattern must contain at least one symbol. The second constraint ensures that ε can only appear at the end of the pattern. It is used in order to prevent same patterns with ε in different positions to be part of the same solution (such as $\langle a, b, \varepsilon \rangle$ and $\langle a, \varepsilon, b \rangle$). Finally, the last constraint states that a pattern must occur at least θ times in the sequence. The goal is then to find an assignment of each P_i satisfying the three constraints. The `episodeSupport` constraint filters from the domains of variables P the infrequent symbols in s at each step in order to find an assignment representing a frequent pattern according to θ. All the inconsistent values of the next uninstantiated variables in the pattern are then removed. Assuming the pattern variables are labeled in static order from left to right, the search is failure free when only this constraint must hold (i.e. all the leaf nodes are solution). Besides, `episodeSupport` is domain consistent: the remaining values in the domain of each variable are part of a solution because all of them have, at least, one support. Additional constraints can also be integrated to the model in order to define properties that the patterns must satisfy. For instance, we can enforce patterns to have at most k symbols or to follow a regular expression.

2.3 Filtering Algorithm

Preprocessing The index of the last position of each symbol in s is stored into a map (*lastPos*). For instance, $s = \langle a, b, a, c, \mathbf{b}, \mathbf{a}, \mathbf{c} \rangle$ gives $\big\{(c \rightarrow 7), (a \rightarrow 6), (b \rightarrow 5)\big\}$. The map can be iterated in a decreasing order by the last positions.


(a) Sequence projection and its reversible vector.

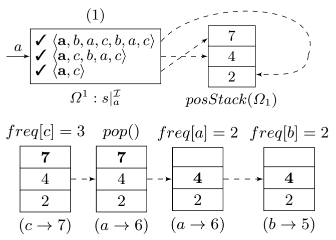

(b) Efficient frequency computation.

Fig. 1: Sequence projection (✓ indicates a match, ✗ otherwise), its reversible vector and frequency computation mechanism.

Sequence Projection and Pseudo Projection The key idea is to successively compute a projection from the previous one each time a variable has been assigned. The assignment of the first variable of the pattern (P_1) involves an *initial projection*. It splits s into a list of subsequences such that each one begins with the projected symbol. The assignment of the other variables (P_2 to P_n) implies an *internal projection*. This behavior is illustrated in the upper part of Fig. 1a for an arbitrary example. The steps leading to pattern $\langle a, b, c, c\rangle$ are detailed. Three subsequences are obtained after an *initial projection* of symbol a $\big((0) \rightarrow (1)\big)$. While there are non empty sequences, *internal projections* are successively performed $\big((1) \rightarrow (4)\big)$ and the pattern (P) is incrementally built.

In practice, only pointers to the position in each sequence where the prefix has matched are stored. It is the mechanism of *pseudo projection*. As Aoga et al. [2], we implement it with a reversible vector (*posv*) and a trail-based structure (lower part of Fig. 1a). The idea is to use the same vector during all the search inside the propagator, and to only maintain relevant start and stop positions. At each propagator call, three steps are performed. First, the last recorded start and stop positions are taken. Secondly, the propagator records the new information in the vector after the previous stop position. Finally, the new positions are updated in order to retrieve the information added. The reversible vector is then built incrementally. For each projection, the corresponding start index (ϕ) in the vector as well as the number of sequences inside the projection (φ) are stored. In other words, information related to a projection are located between indexes $i \in [\phi, \phi + \varphi)$. Besides, the index of the variable P_i that has been assigned (ψ) is also recorded after each projection step. Before the first assignation ψ is equal to zero. The three variables are implemented as reversible integers. Initially, all the indexes are present in the vector, but all along the pseudo-projections, only the non empty sequences are considered.

Propagation The goal is to compute a projection each time a variable has been assigned to a symbol a. Assignments of variables are done successively from the first variable to the last one. The propagator is then called after each assignment. It is shown in Alg. 1. Initialization of reversible structures is done when $\psi = 0$

(lines 8-9). If the last assigned variable (P_ψ) has been bound to ε, the algorithm enforces all the next variables to be also bound to ε (lines 10-12). The pattern is then completed and the propagation is finished. Otherwise, after each variable binding, the projected sequence and the *projected frequencies* are computed (line 14). Finally, all the infrequent symbols are removed from the domain of $P_{\psi+1}$ (lines 15-17). *Projected frequency* of each symbol in the domain of $P_{\psi+1}$ (except ε) is compared to the threshold and removed if it is infrequent.

Sequence Projection Let us now present how sequences are projected (Alg. 2). First, the *projected frequency* of each symbol for the current pseudo projection is set to 0 (*freq* on line 8). The main loop (lines 10-23) iterates over the previous projection thanks to the reversible integers ϕ and φ. At each iteration a value in *posv* is considered. The next condition (line 12) is used to distinguish the *initial* from an *internal projection*. If a is the first projected symbol and if it does not match with the first symbol of the sequence, then the sequence is not included in the projection. Otherwise, an *internal projection* is applied.

The next expression (lines 13-14) is an optimization we introduced, called *early projection stopping*. This optimization is based on one invariant of our structure: it stores suffixes of s with a decreasing order by their size. Each suffix in a projection is then strictly included in all the previous ones. When a no-match has been detected in a sequence, all the next ones can be directly discarded without being checked. It stops the internal projection as soon as possible. The *early projection stopping* gains importance when the number of sequences is large. Then, if a is not present in the current considered sequence, the loop can be stopped and unnecessary computation is avoided.

At this step, we are sure that a appears at least once in the current sequence in the projection. Lines 16 to 21 make the search for the match, either by *position caching*, or by iteration on the positions. *Position caching* is a second optimization we introduced. Once a match has been detected in a sequence, the position of the match is recorded. Thanks to the aforementioned invariant, we are sure that the match in the next sequence cannot occur before this position. If this position is greater than the start position of the sequence (in *posv*), a match is directly detected. The reversible vectors are then updated (line 23). Variable *sup* is used to store the size of the new projection.

The last loop (lines 24-28) updates the *projected frequency* of each symbol. The *projected frequency* of a symbol in a projection corresponds to the number of sequences of the projection beginning at an index lower than the index of the last position of the symbol. This idea was introduced in LAPIN [33] and exploited by Aoga et al. [2]. It can be implemented efficiently thanks to the invariant and *lastPos* map. It is illustrated in Fig. 1b (upper part) with $lastPos = \{(c \to 7), (a \to 6), (b \to 5)\}$. The position just after each match is pushed in a LIFO structure, *posStack*. The last matched position is located on the top of the stack.

Once the stack is obtained, the idea is to successively compare in a decreasing order the last position of each symbol with the top of the stack. Illustration of this behavior is presented in Fig. 1b (lower part). If the last position of a symbol is greater than the top of the stack, it indicates that the symbol occurs at least

Algorithm 1: $propagate(s, \Sigma, a, P)$

```
▷ Internal State: $posv$, $\phi$, $\varphi$ and $\psi$.
▷ Pre: $s$ is the initial long sequence of size $n$.
▷        $\Sigma$ is the set of symbols and $a$ is a symbol.
▷        If $\psi > 0$ then $\langle P_1, \ldots, P_\psi \rangle \in P$ are bound and $P_\psi$ is assigned to $a$.
▷        $\theta$ is the support threshold.
▷        $posv$, $\phi$, $\varphi$ and $\psi$ are the reversible structures as defined before.

if $\psi = 0$ then
    $\phi := 1$   $\varphi := n$   $\psi := 1$   $posv[i] := i$        $i \in [1, n]$
if $P_\psi = \varepsilon$ then
    for $j \in [\psi + 1, n]$ do
        $P_j.assign(\varepsilon)$
else
    $freq := sequenceProjection(s, \Sigma, a)$        ▷ Detailed in Alg. 2.
    foreach $b \in Domain(P_{\psi+1})$ do
        if $b \neq \varepsilon \wedge freq[b] < \theta$ then
            $P_{\psi+1}.removeValue(b)$
```

Algorithm 2: $sequenceProjection(s, \Sigma, a)$

```
▷ Internal State: $posv$, $\phi$, $\varphi$ and $\psi$.
▷ Pre: $s$ is the initial long sequence.
▷        $\Sigma$ is the set of symbols.
▷        $a$ is the current projected symbol ($a = P_\psi$).
▷        $posv$, $\phi$, $\varphi$ and $\psi$ are reversible structures as defined before.
▷        $posv[i]$ with $i \in [\phi, \phi + \varphi)$ is initialized.

$j := \varphi$   $sup := 0$   $prevPos := -1$   $freq[b] := 0 \; \forall b \in \Sigma$
$posStack := Stack()$
for $i \in [\phi, \phi + \varphi - 1]$ do
    $pos := posv[i]$
    if $\psi > 1 \vee a = s[pos]$ then
        if $pos > lastPos[a]$ then
            break                                   ▷ Early projection stopping
        else
            if $prevPos < pos$ then
                while $a \neq s[pos]$ do
                    $pos := pos + 1$
                $prevPos := pos$                    ▷ Position caching
            else
                $pos := prevPos$
            $posStack.push(pos + 1)$
            $posv[j] := pos + 1$   $j := j + 1$   $sup := sup + 1$
foreach $(x, pos_x)$ in $lastPos$ do
    while $posStack.notEmpty \wedge posStack.top > pos_x$ do
        $posStack.pop$
    $freq[x] := posStack.size$                      ▷ Projected frequency
    if $posStack.isEmpty$ then break
$\phi := \phi + \varphi$   $\varphi := sup$   $\psi := \psi + 1$
return $freq$
```

once in the current sequence and consequently in all the previous ones in the stack. The *projected frequency* of this symbol corresponds then to the remaining size of the stack and the next symbol in *lastPos* can be processed. Otherwise, we are sure that the symbol has no occurrence in the current sequence. Its position is popped and the comparison is done with the new top. The resulting *projected frequencies* are $c = 3$, $a = 2$ and $b = 2$. This mechanism has a time complexity of $\mathcal{O}(n + |\Sigma|)$. For comparison, *projected frequencies* are computed in $\mathcal{O}(n \times |\Sigma|)$ by Aoga et al. [2] (each subsequence is scanned at each projection). Finally, the reversible integers are updated and the *projected frequency* map is returned.

Time and Spatial Complexity Main loop of Alg. 2 (lines 10-23) is computed in $\mathcal{O}(n^2)$ and the *projected frequencies* (lines 24-28) in $\mathcal{O}(n + |\Sigma|) = \mathcal{O}(n)$ (because the number of different symbols is bounded by the sequence size). In Alg. 1, lines 8-9 cost $\mathcal{O}(n)$ and the domain pruning (lines 15-17) is performed in $\mathcal{O}(|\Sigma|)$. It gives $\mathcal{O}(n + (n^2 + n) + |\Sigma|) = \mathcal{O}(n^2)$. For the spatial complexity, we have $\mathcal{O}(n + n \times d) = \mathcal{O}(n \times d)$ with d the maximum depth of the search tree, which is the maximum size of the reversible vector. For comparison, an explicit decomposition of the problem gives $\mathcal{O}(n^2 + n \times d) = \mathcal{O}(n^2)$.

3 Mining Episodes in a Timed Sequence

3.1 Technical Background

So far, `episodeSupport` can only deal with *sequences of symbols* where time is not considered. In practice, sequences can also be time-stamped. Such sequences are most often referred as *sequences of events* instead of *sequences of symbols* and new constraints can then be expressed. For instance, we can be interested in finding episodes such that the elapsed time between two events does not exceed one hour. We define a sequence of events $s = \big\langle (s_1, t_1), \ldots, (s_n, t_n) \big\rangle$ over Σ as an ordered list of events (s_i) occurred at time t_i such that for all $i \in [1, n]$ we have $s_i \in \Sigma$ and $t_1 \leq t_2 \leq \ldots \leq t_n$. The list containing only the events is denoted by s^s and the list of timestamps by s^t. All the principles defined in the previous sections are reused. Besides, we are now able to enforce time restrictions. Two of them are used in practice: *gap* and *span*. The former (*gap*) restricts the time between two consecutive events while the latter (*span*) restricts the time between the first and the last event. Considering such restrictions cannot be done only by imposing additional constraints in the model [3]. It requires to adapt the subsequence relations (Def. 9) and to design a dedicated propagator. The concept of *extension window* is also defined. The extension window of an embedding contains only events whose timing satisfies *gap* constraint.

Definition 9 (Subsequence under gap/span). *$\alpha = \langle \alpha_1, \ldots, \alpha_m \rangle$ is a subsequence of $s = \big\langle (s_1, t_1), \ldots, (s_n, t_n) \big\rangle$ under $gap[M, N]$, denoted by $\alpha \preceq^{gap[M,N]} s$, if and only if s^s is a subsequence of embedding $(e_1, \ldots, e_k)$ according to Def. 1, and if $\forall i \in [2, k]$ we have $M \leq t_{e_i} - t_{e_{i-1}} \leq N$. The embedding $(e_1, \ldots, e_k)$ under $\preceq^{gap[M,N]}$ relation is called a $gap[M, N]$-embedding. $(e_1, \ldots, e_k)$ is an* *episode-embedding of α according to Def. 2 where $\preceq^{gap[M,N]}$ is considered for the subsequence relation. The support of α, denoted by $\sigma_s^{gap[M,N]}(e)$, is the number of gap$[M,N]$-embeddings of α in s. Similarly, α is a subsequence of s under span$[W,Y]$, denoted by $\alpha \preceq^{span[W,Y]} s$, if and only if s^s is a subsequence of embedding $(e_1,\dots,e_k)$ according to Def. 1, and if $W \leq t_{e_k} - t_{e_1} \leq Y$. Relation $\preceq^{span[W,Y]}$ and $\sigma_s^{span[M,N]}(e)$ are also defined similarly.*

Example 8. Let us consider $s = \big\langle (a,2),(b,4),(a,5),(c,7),(b,8),(a,9),(c,12)\big\rangle$. $\langle a,b,c\rangle$ is a subsequence of s under *gap*$[1,3]$ with embedding $(1,2,4)$. $(3,5,7)$ is not a *gap*$[1,3]$-embedding because $t_{e_3} - t_{e_2} = 12 - 8 > 3$. Besides, $\langle a,b,c\rangle$ is a subsequence of s under *span*$[6,10]$ with embedding $(3,5,7)$. $(1,2,4)$ is not valid because $t_{e_3} - t_{e_1} = 7 - 2 < 6$.

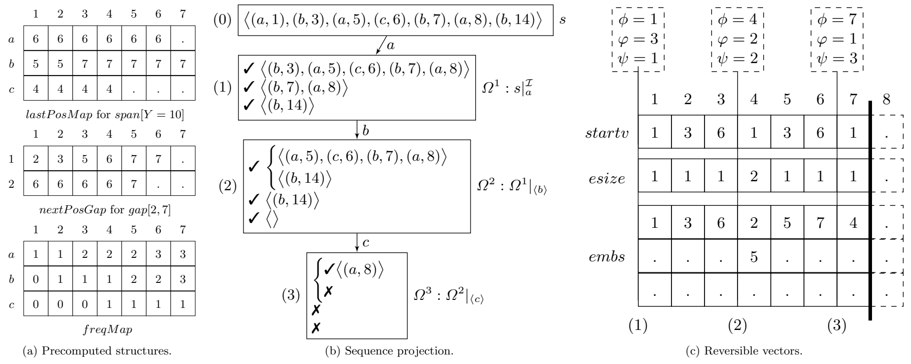
Definition 10 (Extension window). *Let $e = (e_1, e_2, \dots, e_k)$ be any gap$[M,N]$-embedding of a subsequence α in a sequence s. The extension window of this embedding, denoted $\mathrm{ew}_e^{gap[M,N]}(s)$, is the subsequence $\big\langle (s_u,t_u),\dots,(s_v,t_v)\big\rangle$ such that $(t_{e_k} + M \leq t_u) \wedge (t_v \leq t_{e_k} + N) \wedge (t_{u-1} < t_{e_k} + M) \wedge (t_{v+1} > t_{e_k} + N)$. Each embedding has a unique extension window, which can be empty.*

Example 9. Let $(3,4)$ be a *gap*$[2,6]$-embedding of $\langle a,c\rangle$ in sequence s (Example 8). We have $\mathrm{ew}_e^{gap[2,6]}(s) = \big\langle (a,9),(c,12)\big\rangle$.

The goal is to find the all patterns having a support, possibly under *gap* and *span*, greater than the threshold. Let $P = \langle P_1,\dots,P_n\rangle$ be a sequence of variables representing a pattern. the timed version of `episodeSupport`(P,s,θ,M,N,W,Y) enforces the four following constraints: (1) $P_1 \neq \varepsilon$, (2) $P_i = \varepsilon \rightarrow P_{i+1} = \varepsilon$, $\forall i \in \big[1,n\big)$, (3) $\sigma_s^{gap[M,N]}(P) \geq \theta$ and (4) $\sigma_s^{span[M,N]}(P) \geq \theta$.

3.2 Filtering Algorithm

Precomputed Structures The three structures are shown in Fig. 2a. First, the *lastPos* map is adapted from the previous section in order to store the last position of each event that can be matched while satisfying the maximum span (Y). The last position of each event inside each range $[t, t+Y]$ is recorded, where t is the timestamp of the event. Maximum span is then implicitly handled by this structure, which is not done by Aoga et al. [3]. Besides, for each position i in s, the index of the first (u) and the last (v) positions after i such that $t_u \geq t_i + M$ and $t_v \geq t_i + N$ are stored into a map (*nextPosGap*), where M and N are the minimum and maximum gap. The *nextPosGap* is used after each projection in order to directly access the next extension window. Finally, for each position i in s, the number of times that each event has occurred inside the range $[1,i]$ in s is stored (*freqMap*). It is used in order to efficiently compute the *projected frequency* of each symbol during a projection. We can be sure that an event a appears at least once in a window of range $[u,v]$ if the occurrence of a at the end of the window is strictly greater than the occurence of a just before the window ($freqMap[v][a] > freqMap[u-1][a]$). It has not been used by Aoga et al. [3].


(0) $\langle(a,1),(b,3),(a,5),(c,6),(b,7),(a,8),(b,14)\rangle$ s
a
(1) ✓ $\langle(b,3),(a,5),(c,6),(b,7),(a,8)\rangle$
✓ $\langle(b,7),(a,8)\rangle$
✓ $\langle(b,14)\rangle$
$\Omega^1 : s|_a^{\mathcal{I}}$
b
(2) ✓ $\langle(a,5),(c,6),(b,7),(a,8)\rangle$
$\langle(b,14)\rangle$
✓ $\langle(b,14)\rangle$
✓ $\langle\rangle$
$\Omega^2 : \Omega^1|_{\langle b\rangle}$
c
(3) ✓ $\langle(a,8)\rangle$
✗
✗
✗
$\Omega^3 : \Omega^2|_{\langle c\rangle}$
$\phi = 1$ $\varphi = 3$ $\psi = 1$
$\phi = 4$ $\varphi = 2$ $\psi = 2$
$\phi = 7$ $\varphi = 1$ $\psi = 3$
(1)
(2)
(3)

	1	2	3	4	5	6	7
a	6	6	6	6	6	6	.
b	5	5	7	7	7	7	7
c	4	4	4	4	.	.	.

lastPosMap for *span*$[Y = 10]$

	1	2	3	4	5	6	7
1	2	3	5	6	7	7	.
2	6	6	6	6	7	.	.

nextPosGap for *gap*$[2, 7]$

	1	2	3	4	5	6	7
a	1	1	2	2	2	3	3
b	0	1	1	1	2	2	3
c	0	0	0	1	1	1	1

freqMap

(a) Precomputed structures.

(b) Sequence projection.

	1	2	3	4	5	6	7	8
startv	1	3	6	1	3	6	1	.
esize	1	1	1	2	1	1	1	.
	1	3	6	2	5	7	4	.
embs	.	.	.	5	.	.	.	.
	.	.	.	.	.	.	.	.

(c) Reversible vectors.

Fig. 2: Data structures used for timed sequences with *gap*$[2, 7]$ and *span*$[1, 10]$.

Storing Several Embeddings When a *gap* constraint is considered, the anti-monotonicity property does not hold anymore [3]. The main consequence is that all the possible embeddings must be considered, and not only the first one. The projection mechanism (described in Fig. 1a) has then to be adapted. It is illustrated in Fig. 2b. For instance, two embeddings are considered for the projection from (1) to (2) of the first sequence. It is required to record all of the corresponding extension windows in order to miss none supporting event. To do so, a reversible vector (*startv*) recording the start index in s for each sequence is used (Fig. 2c). Besides, other reversible vectors are added: *esize*, which represents the number of embeddings related at each projected sequence and *embs*, which records the start index of the different embeddings. It is a simplified adaptation of the structure proposed by Aoga et al. [3].

Minimum Span The minimum span is not anti-monotonic. Therefore, we do not consider it during the projection but *a posteriori*: it is only checked when a complete pattern is obtained and not before. It requires slight modifications in the *propagate* method (Alg. 1). A variable $\gamma_s(P)$ representing the number of supports satisfying the minimum span constraint for P is recorded and computed during the projection. Once the projections are completely done for this episode (after the line 12), $\gamma_s(P)$ is compared with the support threshold and an *inconsistency* is raised if it is below the threshold.

Sequence Projection Projection mechanism is presented in Alg. 3. Initially, the *projected frequencies* of each event is set to 0 (line 9). When $\psi = 1$, an initial projection is performed (lines 10-20) and we are looking for events that match with a (line 13). Once a match is detected, reversible vectors are updated (line 14). *Projected frequencies* are computed using *nextPosGap* and *freqMap* structures (lines 15-20). First, the window where the events must be considered is computed. Secondly, the *projected frequency* of each event appearing in the window is incremented. When $\psi > 1$, we have an internal projection (lines 21-43). Each embedding is successively considered (line 26). For each one, the sequence is iterated from the first next position satisfying the minimum gap to the last one satisfying the maximum gap, or to the last symbol of the sequence (line 28). Once a match has been detected, the number of possible embeddings is incremented and its position is recorded (line 30). If the embedding of the current pattern satisfies the minimum span constraint, γ_s is incremented (lines 31-32). It is used in the *propagate* method as explained before. Then, *projected frequencies* are computed as in the initial projection (lines 33-39).

Time and Spatial Complexity Let us assume k is the maximum length of time window (often $k \ll n$) and d the maximum depth of the tree search ($d \leq k$). Initial projection (lines 11-21) in Alg. 3 is computed in $\mathcal{O}(n \times |\Sigma|)$: the sequence is completely processed and frequencies are computed at each match. Internal projection (lines 22-44) is computed in $\mathcal{O}(n \times |\Sigma| \times k^2)$. It gives $\mathcal{O}(n \times |\Sigma| + n \times |\Sigma| \times k^2) = \mathcal{O}(n \times |\Sigma| \times k^2)$. For the spatial complexity, vectors have a maximum length of $k \times d$ and there are at most k embeddings, which gives $\mathcal{O}(d \times k^2)$.

4 Experimental Results

This section evaluates the performance of `episodeSupport` on different datasets with and without time consideration. Experiments have been realised on a computer with a 2.7 GHz Intel Core i5 64 bits processor and with a RAM of 8 Go using a 64-Bit HotSpot(TM) JVM 1.8 running on Linux Mint 17.3. Execution time is limited to 1800 seconds unless otherwise stated. The algorithms have been implemented in Scala with OscaR solver [31] and memory assessment has been performed with *java Runtime* classes. For the reproducibility of results, the implementation of both constraints is open source and available online.[1] One synthetic and three real-data sets are considered: proteins from Uniprot database [5], UCI Unix dataset [20] and UbiqLog [26, 27].

Our approach is compared with the existing methods. We identified two ways to mine frequent patterns in a sequence. On the one hand, we can resort to a specialized algorithm. To the best of our knowledge, MINEPI+ and EMMA [14] are the state-of-the-art methods for that. On the other hand, we can explicitly split the sequence into a database and then reduce the problem into an SPM problem. Once done, CP-based methods can be used [2, 3, 16, 23]. Our comparisons are based on the approach of Aoga et al. [2, 3] that turns out to be the most efficient. We refer to it as the Decomposed Frequent Episode Mining (DFEM) approach, or DFEMt when time is considered.

Memory Bound Analysis We applied DFEM and `episodeSupport` on synthetic sequences of different sizes with 100 distinct symbols uniformly distributed in order to define what are the largest sequences that can be processed. We observed that with decomposed approaches, sequences greater than 30000 symbols cannot be processed when memory is limited to 8GB. With `episodeSupport` memory is not a bottleneck.

Comparison with Decomposed Approaches Experiments and results with Uniprot and UbiqLog datasets are shown in Fig. 3 and Table 1. The latter

[1] `https://bitbucket.org/projetsJOHN/episodesupport` (also available in [31])

Algorithm 3: *sequenceProjectionTimed*$(s^s, s^t, \Sigma, a, N, W)$

```
▷ Internal State: $startv$, $esize$, $embs$, $\phi$, $\varphi$, $\psi$, $\gamma_s(P_{:\psi})$.
▷ Pre: $s^s$ and $s^t$ are the event/timestamp list of the initial long sequence.
▷         $\Sigma$ is the set of symbols.
▷         $a$ is the current projected symbol ($a = P_\psi$).
▷         $startv[i]$, $esize[i]$ and $embs[i]$ with $i \in [\phi, \phi + \varphi)$ are initialized.
▷         $\gamma_s(P_{:\psi}) = 0$ with $P_{:\psi}$ the episode represented by $\langle P_1, \dots, P_\psi \rangle$.
▷         $N$ and $W$ are the gap max bound and of the span min bound.

$freq[b] := 0 \quad \forall b \in \Sigma$
if $\psi = 1$ then
    $j := 1$
    for $pos \in [1, |s^s|]$ do
        if $s^s[pos] = a$ then
            $startv[j] := pos \quad esize[j] := 1 \quad embs[j][1] := pos \quad j := j + 1$
            $(u, v) := nextPosGap[pos]$                        ▷ Precomputed structure
            if $u \leq |s^s|$ then
                for $b \in Domain(P_{\psi+1})$ do
                    $l := \mathbf{min}(v, |s^s|)$
                    if $freqMap[l][b] > freqMap[u-1][b]$ then
                        $freq[b] := freq[b] + 1$                ▷ Projected frequency
else
    $j := \phi + \varphi$
    for $i \in [\phi, \phi + \varphi - 1]$ do
        $id := startv[i] \quad nEmb := 0 \quad k := 1 \quad v := -1 \quad isIncremented :=$ false
        $isVisited[b] :=$ false $\quad \forall b \in \Sigma$
        while $v < |s^s| \wedge k \leq esize[i]$ do
            $e := embs[i][k] \quad (pos, \_) := nextPosGap[e]$        ▷ 2nd element unused
            while $v < |s^s| \wedge pos \leq lastPosMap[id][a] \wedge s^t[pos] \leq s^t[e] + N$ do
                if $s^s[pos] = a$ then
                    $nEmb := nEmb + 1 \quad embs[j][nEmb] := pos$
                    if not $isIncremented \wedge s^t[pos] - s^t[id] \geq W$ then
                        $isIncremented :=$ true $\quad \gamma_s(P_{:\psi}) := \gamma_s(P_{:\psi}) + 1$
                    $(u, v) := nextPosGap[pos]$                ▷ Precomputed structure
                    if $u \leq |s^s|$ then
                        for $b \in Domain(P_{\psi+1})$ do
                            $l := \mathbf{min}(v, |s^s|)$
                            if $(freqMap[l][b] > freqMap[u-1][b]) \wedge$ not
                              $isVisited[b]$ then
                                $isVisited[b] :=$ true
                                $freq[b] := freq[b] + 1$        ▷ Projected frequency
                $pos := pos + 1$
            $k := k + 1$
        if $nEmb > 0$ then
            $startv[j] := id \quad esize[j] := nEmb \quad j := j + 1$
$\phi := \phi + \varphi \quad \varphi := j - \phi$
return $freq$
```

presents results for different settings while the former shows the performance profiles [8] for both the memory consumption and the computation time.

We can observe that `episodeSupport` outperforms both decomposed approaches in terms of execution time and memory consumption for most of the instances. Both gains become more important when the sequence is large. Besides, decomposed approaches cannot process the largest sequences regarding the time limitation imposed.

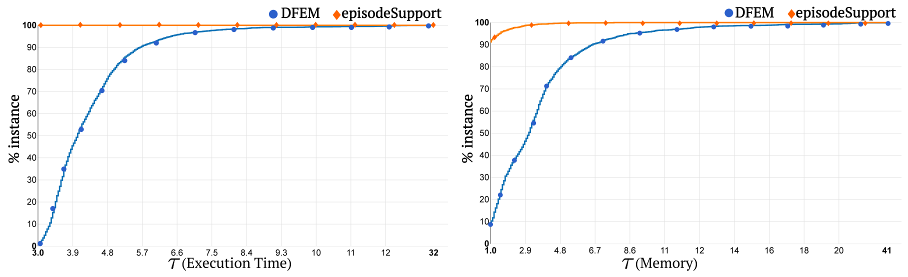
Comparison with Specialized Approaches Experiments on Unix dataset with a threshold of 5% and a maximum span of 10 are provided in [14][2]. Comparisons of these specialized approaches with ours are presented in Table 2. It shows that `episodeSupport` seems competitive with MINEPI+ and EMMA. For the largest sequences (USER8 and USER6), `episodeSupport` is the most efficient. For some instances (USER5 and USER7) that are quickly solved, the cost of initializing the data structures with our approach is higher than the gain obtained. In general, the gain becomes more important when sequences are larger or harder to solve. Finally, given that the implementation of MINEPI+ and EMMA is missing, it is difficult to perform a fair comparison of the approaches.

Handling Additional Constraints Additional constraints can be considered in order to define properties that the patterns must satisfy. No modification of `episodeSupport` is required. Results of experiments are presented in Table 3. The goal was to find frequent episodes ($\theta \geq 20$) having a maximum length of 6, containing at least three `Q` (`atLeast` constraint) but no `D` (`exclusion`), and satisfying the regex `M(A|T).*F` (`regular`). Two episodes (`MTQQQF` and `MAQQQF`) have been discovered. As observed, the additional constraints reduce the execution time as CP takes advantage of the stronger filtering to reduce the search space. This reduction would not be observed with a *generate and filter* approach.

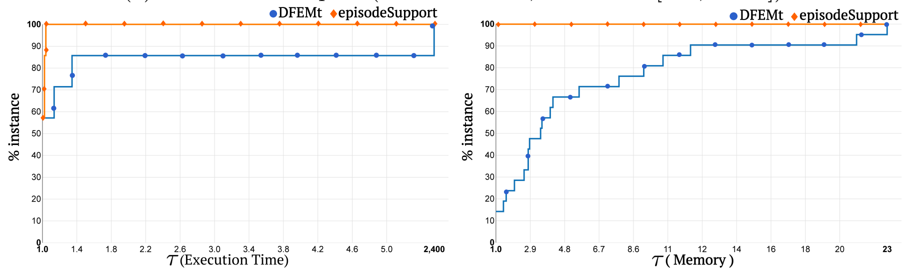
5 Conclusion and Perspective

There is a growing interest for solving data-mining challenges with CP. In addition to the flexibility it brings, recent works have shown that it can provide similar performances, or even better, than specialized algorithms [2, 3]. So far CP has not been considered yet for mining frequent episodes. We introduced two global constraints (`episodeSupport`) for solving this problem with or without time-stamps. It relies on techniques used for SPM such as *pattern growth*, *pseudo projections* and *reversible vectors* but also on new ideas specific to this problem for improving the efficiency of the filtering algorithms (*early projection stopping*, *position caching* and *efficient frequency computation*). Experimental results have shown that our approach provides better performances in terms of execution time and memory consumption than state-of-the-art methods, with the additional benefits that it can accommodate additional constraints.

[2] Results provided in [14] are directly used since the implementation is not available.



(a) Results for Uniprot (2452 instances, where $n \in [100, 30000]$).



(b) Results for Ubiqlog (21 instances, with $gap[100, 3600]$ and $span[1, 35000]$).

Fig. 3: Performance profiles ($\theta = 5\%$, maximum size of 5, timeout of 600 seconds).

Table 1: Execution time and memory usage for several datasets and thresholds.

Patterns having a maximum size of 5							$gap[100, 3600]$ and $span[1, 35000]$						
			Memory (Mb)		Execution time (s)					Memory (Mb)		Execution time (s)	
Name $\|s\| + \|\Sigma\|$	θ	nSol	DFEM	episode Support	DFEM	episode Support	Name $\|s\| + \|\Sigma\|$	θ	nSol	DFEMt	episode Support	DFEMt	episode Support
Q08379 $1002 + 20$	100	437048	45	**34**	9.08	**2.45**	10Mcomplete $1072 + 30$	300	10	107	**35**	0.24	**0.20**
	90	533395	46	**38**	8.68	**1.97**		100	3113	999	**469**	67.83	**29.53**
	70	645834	35	**16**	9.16	**1.67**		50	10481	1226	**505**	118.56	**57.48**
	50	1128537	67	**43**	13.29	**2.59**		20	51108	1110	**599**	204.98	**80.93**
Q54CU4 $11103 + 20$	1110	0	1969	**31**	0.11	**0.02**	9Mcomplete $1128 + 45$	10	6724	244	**139**	**1.00**	1.35
	999	157003	2057	**33**	113.32	**3.49**		8	11626	318	**173**	1.35	**0.71**
	777	1178939	1980	**33**	657.68	**14.90**		6	19340	339	**142**	1.42	**0.86**
	555	1515789	1849	**31**	1414.83	**18.41**		4	23225	349	**138**	1.48	**1.05**
Q9I7U4 $18141 + 20$	1814	336842	6898	**38**	946.54	**20.90**	8Mcomplete $3305 + 44$	300	3734	1888	**506**	197.16	**143.94**
	1632	505263	6426	**40**	1146.22	**24.20**		100	38061	3301	**1179**	745.41	**488.32**
	1269	705640	6819	**21**	1674.80	**22.63**		50	123133	3594	**1496**	1489.18	**740.97**
	907	1515791	6819	**21**	timeout	**52.79**		20	516478	3859	**1309**	timeout	**1163.34**

Table 2: Comparison with MINEPI+ and EMMA ($\theta = 5\%$ and $W = 10$).

Databases Features				Execution Time (s)									
name	$\|s\|$	$\|\Sigma\|$	nSol	MINEPI+	EMMA	episode Support	name	$\|s\|$	$\|\Sigma\|$	nSol	MINEPI+	EMMA	episode Support
USER3	16866	273	46	13.2	0.4	**0.173**	USER5	34821	563	37	4.8	**0.3**	0.724
USER7	17329	449	25	0.6	**0.2**	0.563	USER4	37817	479	48	165.3	1.3	**0.636**
USER2	18738	310	38	43.3	0.6	**0.259**	USER8	54042	706	40	1362.3	9.8	**2.214**
USER1	19881	288	57	93.7	1.2	**0.232**	USER6	64152	609	68	2853.3	14.6	**2.178**

Table 3: Additional constraints on Q08379 Protein (Uniprot).

`Only episodeSupport`		+ `exclusion`		+ `atLeast`		+ `regular`	
nSol: 46221933	time(s): 83.2	nSol: 33388768	time(s): 62.6	nSol: 104536	time(s): 0.642	nSol: 2	time(s): 0.002

References

1. Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., Verkamo, A.I., et al.: Fast discovery of association rules. Advances in knowledge discovery and data mining 12(1), 307–328 (1996)
2. Aoga, J.O.R., Guns, T., Schaus, P.: An efficient algorithm for mining frequent sequence with constraint programming. In: Frasconi, P., Landwehr, N., Manco, G., Vreeken, J. (eds.) Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2016, Riva del Garda, Italy, September 19-23, 2016, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9852, pp. 315–330. Springer (2016), `https://doi.org/10.1007/978-3-319-46227-1`
3. Aoga, J.O.R., Guns, T., Schaus, P.: Mining time-constrained sequential patterns with constraint programming. Constraints 22(4), 548–570 (2017)
4. Calders, T., Dexters, N., Goethals, B.: Mining frequent itemsets in a stream. In: Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on. pp. 83–92. IEEE (2007)
5. Consortium, U., et al.: The universal protein resource (uniprot). Nucleic acids research 36(suppl 1), D190–D195 (2008)
6. Cule, B., Goethals, B., Robardet, C.: A new constraint for mining sets in sequences. In: Proceedings of the 2009 SIAM International Conference on Data Mining. pp. 317–328. SIAM (2009)
7. Das, G., Lin, K.I., Mannila, H., Renganathan, G., Smyth, P.: Rule discovery from time series. In: KDD. vol. 98, pp. 16–22 (1998)
8. Dolan, E.D., Moré, J.J.: Benchmarking optimization software with performance profiles. Math. Program. 91(2), 201–213 (2002), `https://doi.org/10.1007/s101070100263`
9. Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., Hutter, F.: Efficient and robust automated machine learning. In: Cortes, C., Lawrence, N.D., Lee, D.D., Sugiyama, M., Garnett, R. (eds.) Advances in Neural Information Processing Systems 28, pp. 2962–2970. Curran Associates, Inc. (2015), `http://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning.pdf`
10. Ghahramani, Z.: Automating machine learning. Lecture Notes in Computer Science 9852 (2016)
11. Guns, T., Dries, A., Tack, G., Nijssen, S., De Raedt, L.: Miningzinc: A modeling language for constraint-based mining. In: Proceedings of the Twenty-Third international joint conference on Artificial Intelligence. pp. 1365–1372. AAAI Press (2013)
12. Guns, T., Nijssen, S., De Raedt, L.: Itemset mining: A constraint programming perspective. Artificial Intelligence 175(12-13), 1951–1983 (2011)
13. Han, J., Pei, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U., Hsu, M.: Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In: proceedings of the 17th international conference on data engineering. pp. 215–224 (2001)
14. Huang, K.Y., Chang, C.H.: Efficient mining of frequent episodes from complex sequences. Information Systems 33(1), 96–114 (2008)
15. Iwanuma, K., Takano, Y., Nabeshima, H.: On anti-monotone frequency measures for extracting sequential patterns from a single very-long data sequence. In: Cybernetics and Intelligent Systems, 2004 IEEE Conference on. vol. 1, pp. 213–217. IEEE (2004)

16. Kemmar, A., Loudni, S., Lebbah, Y., Boizumault, P., Charnois, T.: A global constraint for mining sequential patterns with gap constraint. In: International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems. pp. 198–215. Springer (2016)
17. Kotthoff, L., Nanni, M., Guidotti, R., OSullivan, B.: Find your way back: Mobility profile mining with constraints. In: International Conference on Principles and Practice of Constraint Programming. pp. 638–653. Springer (2015)
18. Kotthoff, L., Thornton, C., Hoos, H.H., Hutter, F., Leyton-Brown, K.: Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. Journal of Machine Learning Research 17, 1–5 (2017)
19. Laxman, S., Sastry, P., Unnikrishnan, K.: A fast algorithm for finding frequent episodes in event streams. In: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 410–419. ACM (2007)
20. Lichman, M.: UCI machine learning repository (2013), `https://archive.ics.uci.edu/ml/datasets/UNIX+User+Data`
21. Mannila, H., Toivonen, H.: Discovering generalized episodes using minimal occurrences. In: KDD. vol. 96, pp. 146–151 (1996)
22. Mannila, H., Toivonen, H., Verkamo, A.I.: Discovering frequent episodes in sequences extended abstract. In: 1st Conference on Knowledge Discovery and Data Mining (1995)
23. Negrevergne, B., Guns, T.: Constraint-based sequence mining using constraint programming. In: International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems. pp. 288–305. Springer (2015)
24. Nijssen, S., Guns, T.: Integrating constraint programming and itemset mining. Machine Learning and Knowledge Discovery in Databases pp. 467–482 (2010)
25. Pesant, G.: A regular language membership constraint for finite sequences of variables. In: International conference on principles and practice of constraint programming. pp. 482–495. Springer (2004)
26. Rawassizadeh, R., Momeni, E., Dobbins, C., Mirza-Babaei, P., Rahnamoun, R.: Lesson learned from collecting quantified self information via mobile and wearable devices. Journal of Sensor and Actuator Networks 4(4), 315–335 (2015)
27. Rawassizadeh, R., Tomitsch, M., Wac, K., Tjoa, A.M.: Ubiqlog: a generic mobile phone-based life-log framework. Personal and ubiquitous computing 17(4), 621–637 (2013)
28. Schaus, P., Aoga, J.O.R., Guns, T.: Coversize: A global constraint for frequency-based itemset mining. In: Beck, J.C. (ed.) Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10416, pp. 529–546. Springer (2017), `https://doi.org/10.1007/978-3-319-66158-2_34`
29. Shokoohi-Yekta, M., Chen, Y., Campana, B., Hu, B., Zakaria, J., Keogh, E.: Discovery of meaningful rules in time series. In: Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining. pp. 1085–1094. ACM (2015)
30. Tatti, N., Cule, B.: Mining closed strict episodes. In: Data Mining (ICDM), 2010 IEEE 10th International Conference on. pp. 501–510. IEEE (2010)
31. Team, O.: OscaR: Scala in OR (2012)
32. Yang, Q., Wu, X.: 10 challenging problems in data mining research. International Journal of Information Technology & Decision Making 5(04), 597–604 (2006)

33. Yang, Z., Wang, Y., Kitsuregawa, M.: Lapin: effective sequential pattern mining algorithms by last position induction for dense databases. Advances in Databases: Concepts, Systems and Applications pp. 1020–1023 (2007)
34. Zhou, W., Liu, H., Cheng, H.: Mining closed episodes from event sequences efficiently. In: Pacific-Asia Conference on Knowledge Discovery and Data Mining. pp. 310–318. Springer (2010)