Noname manuscript No.
(will be inserted by the editor)

Mining Time-constrained Sequential Patterns with Constraint Programming

John O.R. Aoga · Tias Guns · Pierre Schaus

Received: date / Accepted: date

Abstract Constraint Programming has proven to be an effective platform for constraint based sequence mining. Previous work has focussed on standard frequent sequence mining, as well as frequent sequence mining with a maximum 'gap' between two matching events in a sequence. The main challenge in the latter is that this constraint can not be imposed independently of the omnipresent frequency constraint. Indeed, the gap constraint changes when a subsequence is included in a sequence. In this work, we go beyond that and investigate the integration of timed events, the gap constraint as well as the span constraint that constrains the time between the first and last matching event. We show how the three are interrelated, and what the required changes to the frequency constraint are. Key in our approach is the concept of an extension window defined by gap/span and we develop techniques to avoid scanning the sequences needlessly, as well as using a backtracking-aware datastructure. Experiments demonstrate that the proposed approach outperforms both specialized and CP-based approaches in most cases and that the difference becomes increasingly large for low frequency thresholds.

Keywords Data mining · Sequential pattern mining · Constraint programming · Global constraint · Gap constraint · Span constraint · Time constraint

J.O.R Aoga (✉)
Institute of Information and Communication Technologies, Electronics and Applied Mathmatics (ICTEAM), UCLouvain, Belgium
Ecole Doctorale Science de l'Ingénieur (ED-SDI), Université d'Abomey-Calavi (UAC), Bénin
Orcid Id: 0000-0002-7213-146X
E-mail: john.aoga@{uclouvain.be,gmail.com}

T. Guns
Vrije Universiteit Brussel (VUB), Brussels, Belgium
Katholieke Universiteit Leuven, Belgium
E-mail: tias.guns@{vub.ac.be,cs.kuleuven.be}

P. Schaus
Institute of Information and Communication Technologies, Electronics and Applied Mathmatics (ICTEAM), UCLouvain, Belgium
E-mail: pierre.schaus@uclouvain.be

1 Introduction

Sequential pattern mining (SPM) is an important research domain within data mining and widely used in applications such as the web log mining, disease diagnoses mining, event sequence mining, etc [1]. The problem of SPM is to find frequent sequence patterns (also called sequential patterns) in a database of sequences, i.e. ordered list of events that together occur in the data more than a given number of times. This task is a great challenge since the search space is extremely large; $O(m^n)$ solutions are available for patterns with length at most n and for sequences with an average number of m events.

In practice, finding all sequential patterns is typically not enough, as often an overwhelming number of patterns is returned. Hence, there is a need to guide the search towards patterns of interest to the practitioner. This calls for techniques which can incorporate preferences or restrictions on the length and content of the patterns (constraints). In many applications, the time elapsed between events is also important to take into account.

Assume for instance a database containing sequences of web pages visited by users on a given web-site. One could be interested in access patterns within a single browsing session, for example with no more than 20 minutes time between two pages. Also in biological sequence mining the position and distance of the symbols in the sequence matter. A constraint on the maximum time between any two consecutive symbols in the pattern is called a *gap* constraint, while a constraint on the time from the first to the last event is called a *span* constraint. In this work, we assume all sequences have explicit timestamps and the goal is to support gap and span constraints as well as constraints on the frequency as well as constraints on the syntax of the patterns.

Related work The problem of sequential pattern mining, first introduced by Agrawal et al. [2], is widely studied [2, 13, 14, 28, 34, 36, 37, 38, 39] with many applications as well [16, 17]. These works can be caregorized into *1)* apriori-based (horizontal/vertical formatting)[5, 34, 38] and *2)* projection-based[28] methods. In general, users only need a small subset of the found patterns. Hence, a number of works have focused on the addition of user-defined constraints such as inclusion/exclusion items, pattern length (minimum/maximum), super-pattern, aggregate function (sum, average, maximum, minimum, standard deviation, ...), regular expression and span/gap. They are widely discussed in [29].

GSP [34] was the first approach including gap and span constraints. This method is not very efficient since it requires to generate all candidate patterns and to scan the dataset several times. Some approaches added the constraints in a post-processing step [27]. In the *cSPADE* algorithm [38], the constraints are directly integrated into the frequent pattern search process. It efficiently takes into account constraints such as length and width restrictions on the pattern, item constraints, minimum and maximum gaps between events, as well as a maximum span. Unlike *cSPADE*, *GenPrefixSpan* [3] is an extension of the depth-first *PrefixSpan* [28] algorithm to allow gap constraints. Time constraints on the sequences (instead of events) have also been investigated in [9]. Special classes of SPM problem or constraints was also tackled: the closed/maximal SPM [10, 21, 22, 35, 36] the multi-dimensional SPM [31], the episodes events [23], etc. However, all the above-mentioned approaches lack flexibility at the algorithmic level, since adding a new constraint often involves changing the whole algorithm and may hinder scalability. We are for instance not aware of a tool that can efficiently take regular expression constraints into account together with time constraints.

As an alternative, the use of constraint programming has been investigated [4, 8, 12, 18, 19, 20, 24, 25]. Kemmar et al. [20] have subsequently shown that this approach can be made more scalable by grouping all low-level constraints involving the frequency computation into one global constraint. Moreover, they investigated the top-k sequential pattern mining problem [18]. More recently, we have shown that combining this approach with algorithmic techniques from both the CP community and the data mining community can result in a global constraint that outperforms generic as well as specialized methods [4].

While the above CP methods can handle constraints on the pattern syntax, gap and span constraints are only supported by the much less efficient approach of [25]. The reason is that the timing information is hidden in the global *frequency* constraint. Hence, Kemmar et al. [19] extended their global constraint for the gap constraint specifically.

Contribution In this work, we wish to improve on [19] and [4] by modifying the global frequency constraint to capture the most common time-related constraints: explicitly timed events, minimum/maximum gap, and minimum/maximum span. To maintain scalability, we must ensure that we do not needlessly scan the sequences in the database during the search. Our contributions can be summarized as follows: *1)* we adapt the backtracking-aware datastructure introduced in [4] to store all possible occurrences of the pattern in a sequence, including the first matching symbol to support *span* constraints; *2)* we avoid scanning a sequence for a symbol beyond the (precomputed) last occurrence of that symbol in the sequence; *3)* we introduce the concept of *extension window* of an embedding and avoid to scan overlapping windows multiple times; *4)* we avoid scanning for the start of an extension window, which is specific to the minimum gap constraint, by precomputing these in advance; and finally *5)* we experimentally show that using this global constraint we outperform other sequence mining algorithms in all but a few cases. Furthermore, this time-aware global constraint can be combined with other constraints string constraints [15] such as regular expression [30], grammar [32], item inclusion/exclusion or pattern length constraints.

2 Preliminaries

In this section, we revisit the preliminary concepts for both Sequential Pattern Mining (SPM) and Constraint Programming (CP). Most of these concepts can be found for SPM in [1, 40] and in [33] for CP.

2.1 Sequential Pattern Mining Background

Assume $\mathcal{I} = \{1, \dots, L\}$ is an alphabet, that is, a list of possible symbols. Table 1a represents a sequence database (SDB) with timestamps. The database is a set of tuples (sid, s) where sid is the sequence identifier and $s = \langle (s_1, t_1)(s_2, t_2) \dots (s_n, t_n) \rangle$ is a sequence; an ordered list of symbols/events (s_k) occurred at time t_k, where

a) Sequence database (SDB)		b) *nextPosGap*							c) *lastPosMap*				
sid	sequence	1	2	3	4	5	6	7	A	B	C	D	E
sid_1	$\langle (A,2)(B,5)(D,6)(C,10)(B,11) \rangle$	2	4	4	6	6			1	5	4	3	0
sid_2	$\langle (B,1)(A,2)(A,9)(D,12)(C,15)(A,18)(B,24) \rangle$	3	3	4	5	6	7	8	6	7	5	4	0
sid_3	$\langle (A,2)(B,4)(D,6)(D,8)(B,10)(E,12)(C,14) \rangle$	3	4	5	6	7	8	8	1	5	7	4	6
sid_4	$\langle (A,1)(C,2)(C,3)(B,4) \rangle$	4	5	5	5				1	4	3	0	0

Table 1: a) A sequence database SDB, **b)** a structure for the next position of minimum gap time N(precomputed) and **c)** the last position map.

$t_1 \leq t_2 \leq \ldots \leq t_n$. We use s_i^s, respectively s_i^t, to represent just the list of symbols, respectively timestamps, of sequence i. In the rest of the paper, we assume sequence databases have timestamps, and when the exact timing is not important we will write $\langle ABC \rangle$ to mean $\langle (A,1)(B,2)(C,3) \rangle$.

Example 1 $s = \langle (A,2)(B,4)(D,6)(D,8)(B,10)(E,12)(C,14) \rangle$ is a sequence, $s^s = \{A, B, D, D, B, E, C\}$, $s^t = \{2, 4, 6, 8, 10, 12, 14\}$ and its length $size(s) = 7$.

A sequence can be a subsequence of another sequence. For example $\alpha = \langle ADC \rangle$ is a subsequence of s. More formally, the subsequence relation is:

Definition 1 Subsequence relation ($\preceq$). $\alpha = \langle \alpha_1 \alpha_2 \ldots \alpha_k \rangle$ is a subsequence of $s = \langle (s_1, t_1)(s_2, t_2) \ldots (s_n, t_n) \rangle$ denoted by $\alpha \preceq s$ iff *(i)* $k \leq n$ and *(ii)* there exists a list of integers $(e_1, \ldots, e_k)$, an embedding, with $1 \leq e_1 \ldots \leq e_k \leq n$ such that $s^s[e_i] = \alpha_i$.

Example 2 $\alpha = \langle ADC \rangle$ is a subsequence s with embedding $(1,3,7)$. Another valid embedding would be $(1,4,7)$. Note that for this standard subsequence relation, timing is not important.

The *coverage* of a pattern in a sequence database is the set of sequences in SDB for which our pattern is a subsequence: $Cover_{SDB}(\alpha) = \{(sid, s) \in SDB \mid \alpha \preceq s\}$. We denote by *frequency* of a sequence the size of its cover ($|Cover_{SDB}(\alpha)|$) and by *support* the relative frequency ($Support_{SDB}(\alpha) = |Cover_{SDB}(\alpha)|/size(SDB)$).

Example 3 Subsequence $\alpha = \langle ADC \rangle$ is a subsequence of sequence 1, 2 and 3 in Table 1a, hence $Cover_{SDB}(\langle ADC \rangle) = \{sid_1, sid_2, sid_3\}$, its frequency is 3 and $Support_{SDB}(\langle ADC \rangle) = 75\%$.

The problem of SPM, first introduced by Agrawal et al. [2], is as follows:

Definition 2 SPM problem. Find all subsequences (α) in SDB such that $Support_{SDB}(\alpha) \geq \theta$ where θ is the given support threshold. Each such subsequence α is called a frequent sequence pattern or simply sequential pattern.

In the remaining of the paper, we will use *sequential pattern* to mean *frequent sequence pattern* and *sequence pattern* if the frequency of the pattern does not matter.

There exist multiple algorithms for the SPM problem. The *PrefixSpan* algorithm [28] is among the most famous ones and relies on the idea of the prefix-projected database. Our approach will build on this concept.

Definition 3 Prefix, prefix-projected database. Let $\alpha = \langle \alpha_1 \ldots \alpha_k \rangle$ be a pattern. If a sequence $\beta = \langle \beta_1 \ldots \beta_n \rangle$ is a super-sequence of α: $\alpha \preceq \beta$, then the *prefix* of β w.r.t. α is the *smallest prefix* of β that is still a super-sequence of α: $\langle \beta_1 \ldots \beta_j \rangle$ s.t.

$\alpha \preceq \langle \beta_1 \dots \beta_j \rangle$ and $\nexists j' < j : \alpha \preceq \langle \beta_1 \dots \beta_{j'} \rangle$. The sequence $\langle \beta_{j+1} \dots \beta_n \rangle = \mathit{suffix}_\alpha(\beta)$ is called the suffix and it represents the prefix-projection obtained by *projecting* the prefix away. A prefix-projected database of a pattern α, denoted by $SDB|_\alpha$, is the set of prefix-projections of all sequences in SDB that are super-sequences of α: $SDB|_\alpha = \{(sid_i, \mathit{suffix}_\alpha(sid_i)) \,|\, \alpha \preceq SDB[sid_i]\}$.

Example 4 Consider our running example in Table 1a, where we omit timing information. Assume $\alpha = \langle A \rangle$, then $SDB|_\alpha = \{(sid_1, \langle BDCB \rangle), (sid_2, \langle ADCAB \rangle), (sid_3, \langle BDDBEC \rangle), (sid_4, \langle CCB \rangle)\}$ (Find more example in Table 2a).

The *prefix-projected frequency* of the symbol $a \in I$ ($\mathit{freqs}(a, SDB|_\alpha)$) is the number of sequences in $SDB|_\alpha$ where this symbol appears in the suffix: $\mathit{freqs}(a, SDB|_\alpha) = |\{(sid, s) \in SDB|_\alpha \,|\, a \in \mathit{suffix}_\alpha(s)\}|$. We use $\mathit{freqs}(a)$ instead of $\mathit{freqs}(a, SDB|_\alpha)$ when no ambiguity is possible about the database. Thus, the prefix-projected frequencies of $SDB|_{\langle A \rangle}$ are: $\mathit{freqs}(A) = 1$, $\mathit{freqs}(B) = 4$, $\mathit{freqs}(C) = 4$, $\mathit{freqs}(D) = 3$, $\mathit{freqs}(E) = 1$.

Performing a depth-first search, the *PrefixSpan* algorithm starts with an empty prefix and extends a pattern with one symbol at each step. Then, it computes the projected database and prefix-projected frequencies and extends it again (using only the prefix-projected frequent items). This process continues until all frequent patterns are found. This method is efficient because it avoids extending patterns with infrequent symbols. Also, instead of storing all suffixes explicitly, it maintains just one pointer to the suffix starting position for each sequence. This is called the *pseudo-projected* database $pSDB|_\alpha = \{(sid_i, j + 1) \in SDB\}$ such that $\langle \beta_{j+1} \dots \beta_n \rangle = \mathit{suffix}_\alpha(sid_i)$.

Example 5 Extending prefix $\langle A \rangle$ with $\langle D \rangle$ over $SDB|_{\langle A \rangle}$ gives $SDB|_{\langle AD \rangle} = \{(sid_1, \langle CB \rangle), (sid_2, \langle CAB \rangle), (sid_3, \langle DBEC \rangle)\}$ and can be represented as the pseudo-projected database: $pSDB|_{\langle A \rangle} = \{(sid_1, 4), (sid_2, 5), (sid_3, 4)\}$.

We now recall the subsequence relation under a $\mathit{gap}^{[M,N]}$ constraints, with M the minimum and N the maximum gap between two subsequent events, and under a $\mathit{span}^{[W,Y]}$ constraints with W the minimum and Y the maximum span between the first and last event. This requires changing the subsequence definition in Definition 1.

Definition 4 Subsequence relation under *gap* ($\preceq^{gap^{[M,N]}}$). A sequence $\alpha = \langle \alpha_1 \alpha_2 \dots \alpha_k \rangle$ is a subsequence of $s = \langle (s_1, t_1)(s_2, t_2) \dots (s_n, t_n) \rangle$ under $\mathit{gap}^{[M,N]}$ constraint ($\alpha \preceq^{gap^{[M,N]}} s$) iff *(i)* $k \leq n$; *(ii)* there exists a list of integers $(e_1, \dots, e_k)$, an embedding, with $1 \leq e_1 \dots \leq e_k \leq n$ such that $s^s[e_i] = \alpha_i$; and *(iii)* the time between two consecutive events $t_{e_i - 1}$ and t_{e_i} must be between M and N for all $i \in [2, k]$, $M \leq t_{e_i} - t_{e_{i-1}} \leq N$. An embedding $(e_1, \dots, e_k)$ for $\alpha \preceq^{gap^{[M,N]}} s$ is called a $\mathit{gap}^{[M,N]}$-embedding.

We can similarly define $\preceq^{span^{[W,Y]}}$ where condition *(iii)* becomes: the time between the first event t_{e_1} and the last t_{e_k} must be between W and Y i.e. $W \leq t_{e_k} - t_{e_1} \leq Y$. We can similarly define the *gap* + *span* subsequence relation which contains both conditions.

Example 6 Given sid_1 with $s_1^t = \{2, 5, 6, 10, 11\}$ and sid_2 with $s_2^t = \{1, 2, 9, 12, 15, 18, 24\}$ in Table 1a. Then, $\langle ADB \rangle \preceq^{gap^{[3,7]}} SDB[sid_1]$ with embedding $(e_1, e_2, e_3) = (1, 3, 5)$ because $\{3 \leq (s_1^t[e_2] - s_1^t[e_1] = 4) \leq 7$ and $3 \leq (s_1^t[e_3] - s_1^t[e_2] =$

$5) \leq 7\}$. Note the difference between the positions e_i of the embedding and its time $s_1^t[e_i]$. As another example, $\langle ADB\rangle \npreceq^{gap^{[3,7]}} SDB[sid_2]$ because $3 \leq (s_2^t[e_3]-s_2^t[e_2] = 12) \nleq 7$. Similarly, this embedding and hence the sequence respects a $span^{[8,10]}$ constraint in sid_1: $8 \leq s_1^t[e_3] - s_1^t[e_1] = 9 \leq 10$.

The definition of $Cover_{SDB}(\alpha)$, $Support_{SDB}(\alpha)$ and the SPM problem can be easily adapted to use the gap/span subsequence relation instead of the original relation $\preceq$.

Example 7 Assume $\alpha = \langle ADC\rangle$, $\theta = 3$ and $gap^{[3,7]}$, $Cover_{SDB}^{gap^{[3,7]}}(\alpha) = \{sid_1, sid_2, sid_3\}$ and hence $Support_{SDB}^{gap^{[3,7]}}(\alpha) = 3$. Thus, α is a gap-constrained sequential pattern for the given threshold.

In general, the search space to find the sequential patterns is huge. Hence, to reduce this space several algorithms rely on the *anti-monotonicity property*.

Property 1 **(Anti-monotonicity).** Assume C is a constraint. C is anti-monotone if a sequence s satisfies C all its subsequences also satisfy C and reversely if a sequence s doesn't satisfy C all its super-sequences also doesn't.

The minimum $gap(M)$ and the maximum $span(Y)$ are anti-monotone constraints but the maximum gap (N) constraint violates this property.

Example 8 Assuming our running example, $\langle ADC\rangle$ is frequent under $gap^{[3,7]}$ with $\theta = 3$ but $\langle AC\rangle$ is not frequent ($Support_{SDB}^{gap^{[3,7]}}(\langle AC\rangle) = 2 < 3$).

Fortunately, the maximum gap constraint is *prefix anti-monotone* i.e. every prefix of p satisfies the maximum gap constraint if p satisfies it [29]. We use this property to filter infrequent patterns. The minimum $span(W)$ doesn't satisfy those properties, hence we will apply it in post-processing.

3 CP-based model for SPM problem

A constraint satisfaction problem [33] is defined as a triplet (V, D, C) where V is a set of decision variables with their domains D (possible values of V). C is a set of constraints, each constraint is defined over V and restricts the possible values that these variables can take. Solving the problem of SPM using constraint programming (CP) consists of defining the model (V, D, C).

We present *CP model* of sequential pattern mining introduced in [25] and the *GapSeq* [19] and *PPIC (Prefix Projection Incremental Counting)* [4] global constraints.

Definition 5 Variables and Domains for SPM [25]. Let l be the length of the longest sequence in SDB ($l = max(\{size(s) \mid s \in SDB\})$); $P = [P_1, P_2, \ldots, P_l]$ is an array of variables, representing a pattern, where each P_i represents the i^{th} symbol in the pattern. The domain D_i of P_i is the set of symbols $\mathcal{I}$ plus the empty symbol ϵ:$D_i(P_i) = \{\epsilon\} \cup \mathcal{I}$.

Example 9 For instance for the dataset in Table 1a, $l = 7$, $P = [P_1, \ldots, P_7]$ and for all $i \in [1, l]$, $D_i = \{\epsilon, A, B, C, D, E\}$. $\langle A, D, C\rangle$ corresponds to $P = [A, D, C, \epsilon, \epsilon, \epsilon, \epsilon]$.

Definition 6 Filtering rules. Assume $\forall i \leq l$, $p = \langle p_1, \ldots, p_i \rangle$ is the assigned values of variables $\{P_1, \ldots, P_i\}$, a CP model over P represents the SPM problem given a threshold θ, $gap^{[M,N]}$ and $span^{[W,Y]}$ iff the following conditions are satisfied by every valid assignment to P:

1. $P_1 \neq \epsilon$ (to avoid an empty pattern);
2. $\forall i \in \{2, \ldots, l-1\} : P_i = \epsilon \Rightarrow P_{i+1} = \epsilon$ (to allow pattern with $length < l$);
3. Frequency constraint: $Support_{SDB}(p) \geq \theta$;
4. Frequency under $gap^{[M,N]}$ constraint: $Support_{SDB}^{gap^{[M,N]}}(p) \geq \theta$;
5. Frequency under $span^{[W,Y]}$ constraint: $Support_{SDB}^{span^{[W,Y]}}(p) \geq \theta$.

PPIC **[4] global constraint.** $PPIC(P, SDB, \theta)$ is a global constraint for the SPM problem without gap/span, built on prefix-projection principle, which encodes conditions 1,2,3 in a single propagator. It improved on the state-of-the-art with four elements: *(a)* a backtracking-aware datastructure inspired by *trail-based CP technique*, *(b)* efficient support counting by precomputing the last positions of each symbol, *(c)* not scanning sequences whose prefix can not contain the symbol (precomputed) and *(d)* removing the infrequent symbols of the projected database only from the next domain D_{i+1}.

GapSeq **[19] global constraint.** $GapSeq(P, SDB, \theta, M, N)$ is a global constraint for SPM problem under $gap^{[M,N]}$ which encodes conditions 1,2,4 in a single propagator with the limitation that gap constraints are expressed in terms of position distances i.e. the gap are measured according to the number of events hence time doesn't matter.

4 Embedding database and extension windows

In this section, we introduce the notions of *embedding database* and *extension windows* which reconsider the concept of projected database to incorporate time constraints.

In fact, when having a gap constraint and using prefix-projection (see Definition 3), the assumption that a pattern can be extended with the symbol appearing after the *smallest* matching prefix does not hold anymore. That is, given a sequence, if the first embedding of the prefix cannot be extended because the gap is too small or large, there could exist another embedding that can be extended.

Example 10 Assume the pattern $\alpha = \langle ADC \rangle$ and $gap^{[3,7]}$. There are two embeddings of α in sid_3: $(1, 3, 7)$ and $(1, 4, 7)$. The first embedding is not a $gap^{[3,7]}$-embedding since $3 \leq (t_7 - t_3 = 8) \nleq 7)$ while the second one is.

Hence, it is not sufficient to store just the (suffix of the) smallest embedding as is done in the pseudo-projected database. Instead, we can store all embeddings. One can draw the parallel of this notion with the notion the pseudo-projected database $pSDB|_\alpha$ but instead of only storing the first embedding, we store all available embeddings:

Definition 7 Embedding database ($embSDB|_\alpha$). Assume a sequence $s = \langle (s_1, t_1) (s_2, t_2) \ldots (s_n, t_n) \rangle$ and a subsequence $\alpha = \langle \alpha_1 \alpha_2 \ldots \alpha_k \rangle$ with $k \leq n$. The set of all embeddings of α in s is $emb_\alpha(s) = \{(e_1, \ldots, e_k) | 1 \leq e_1 \leq e_k \leq n$ such that $s^s[e_i] = \alpha_i\}$. The embedding database of α is now defined as $embSDB|_\alpha = \{\, (sid, emb_\alpha(s)) \mid (sid, s) \in SDB\}$.

a) Without time constraints: Projected Database (since time is not matter we omit it)

sid	$pSDB\vert_{\langle A\rangle}$	$SDB\vert_{\langle A\rangle}$'s of $pSDB\vert_{\langle A\rangle}$	$pSDB\vert_{\langle AD\rangle}$	$SDB\vert_{\langle AD\rangle}$'s of $pSDB\vert_{\langle AD\rangle}$
sid_1	1	$\langle BDCB\rangle$	3	$\langle CB\rangle$
sid_2	2	$\langle ADCAB\rangle$	4	$\langle CAB\rangle$
sid_3	1	$\langle BDDBEC\rangle$	2	$\langle DDBEC\rangle$
sid_4	1	$\langle CCB\rangle$		

b) Considering time constraints: Embedding database and extension windows

sid	$embSDB\vert^{[3,7]}_{\langle A\rangle}$	$\text{ew}^{gap^{[3,7]}}_e(s)$'s of $embSDB\vert^{[3,7]}_{\langle A\rangle}$	$embSDB\vert^{[3,7]}_{\langle AD\rangle}$	$\text{ew}^{gap^{[3,7]}}_e(s)$'s of $embSDB\vert^{[3,7]}_{\langle AD\rangle}$
sid_1	(1)	$\langle(B,5)(D,6)\rangle$	(1,3)	$\langle(C,10)(B,11)\rangle$
sid_2	(2),(3),(6)	$\langle(A,9)\rangle$,$\langle(D,12)(C,15)\rangle$,$\langle(B,24)\rangle$	(3,4)	$\langle(C,15)(A,18)\rangle$
sid_3	(1)	$\langle(D,6)(D,8)\rangle$	(1,3),(1,4)	$\langle(B,10)(E,12)\rangle$,$\langle(E,12)(C,14)\rangle$
sid_4	(1)	$\langle(B,4)\rangle$		

Table 2: Embedding database and extension windows for patterns $\langle A\rangle$ and $\langle AD\rangle$: **a)** without time constraints **b)** with time constraints ($gap^{[3,7]}$). Note that embeddings are positions in s, not timings and these positions start from 0.

The embedding database under $gap^{[M,N]}$ of a sequence s is the set of transactions identifiers together with all embeddings of s that satisfy the $gap^{[M,N]}$ constraint; denoted as $embSDB|^{[M,N]}_{\alpha}$. Similarly for the embedding database under $span^{[W,Y]}$ and the combination of gap and span.

GapSeq[19] stores for each embedding the position after the last embedding, called *right pattern extensions*, but this is not sufficient to support a *span* constraint. Our method will store the start and stop position of each embedding, which is sufficient for span, gap and the combination of the two.

Given a span and/or gap constraint, an embedding can only be extended with events whose timing satisfies the span/gap constraints. We call this subsequence of events the extension window of an embedding:

Definition 8 Extension Window (ew**).** Assume a given sequence $s = \langle(s_1, t_1)(s_2, t_2)\dots(s_n, t_n)\rangle$, a subsequence $\alpha = \langle\alpha_1\alpha_2\dots\alpha_k\rangle$ and a $gap^{[M,N]}$ constraint. Let $e = (e_1, e_2, \dots, e_k)$ be any valid $gap^{[M,N]}$*-embedding* of α in s. The *extension window* of this embedding, denoted $\text{ew}^{gap^{[M,N]}}_e(s)$, is the subsequence $\langle(s_u, t_u)(s_{u+1}, t_{u+1})\dots(s_{v-1}, t_{v-1})(s_v, t_v)\rangle$ such that $(t_{e_k} + M \leq t_u) \wedge (t_v \leq t_{e_k} + N) \wedge (\nexists t'_u \in s^t, t_{e_k} + M \leq t'_u < t_u) \wedge (\nexists t'_v \in s^t, t_v < t'_v \leq t_{e_k} + N\}$. The start and the end **position** of this extension window are respectively u and v.

Example 11 Assume $gap^{[3,7]}$ and $\alpha = \langle A\rangle$. For sid_3 in Table 1a, there is one $gap^{[3,7]}$-embedding: (1) with extension window $\langle(D,6)(D,8)\rangle$; hence, if $\langle A\rangle$ is extended with any symbol other than D it will no longer be covered. For $\alpha = \langle A, D\rangle$ there are now two possible embeddings: $(1,3)$ and $(1,4)$. Their extension windows are: $\langle(B,10)(E,12)\rangle$,$\langle(E,12)(C,14)\rangle$. Table 2b shows the embeddings and extension windows for these two patterns for all sequences in the SDB of Table 1a. A comparison can be done with the same versions without time restrictions presented in Table 2a.

5 Trail-based datastructures

Trailing as a mechanism to restore the state. CP-Solvers implementing the depth-first search backtracking algorithms need an efficient state restoration system [33].

This system is based on the *trail* and *time-stamping* mechanism[1]. *Trailing* consists of recording the changes in a node to be able to restore it later on backtrack. The main advantage of trailing is that it makes it possible to focus on the design of the filtering only without worrying about the state restoration. In each node, during the fix-point computation, a state can change several times. The trail keeps a time time-stamp associated with a memory location to avoid storing a same memory on the trail more than once per search node. CP-Solvers typically expose some "reversible" objects externally whose use this mechanism. The *reversible integer*, denoted by *rint*, is an example of "reversible" objects for the primitive type *int*. Our trail-based datastructure (see Section 5) also uses this mechanism.

Trailing the embedding database. We introduce a trailed based datastructure to efficiently store and restore the embedding database. The key idea is to store the embedding database in 'backtracking aware' vectors.

This idea was introduced in *PPIC* [4] allowing to drastically speeding up the search for frequent sequential patterns without time constraints. See an illustration of this datastructure based on the projected database examples of the Table 2a in the Fig. 1a. Two reversible integers store respectively the start position in the vector (ϕ) and the number of entries (φ) in the embedding database. When branching, data is appended at the $\phi+\varphi$ position and ϕ and φ are updated. When backtracking, only the start position and number of elements need to be restored/trailed; the vector can stay unchanged in memory, with the parts after $\phi + \varphi$ overwritten later. In [4], only the sequence ids (*sids* vector) and the start position of the suffixes (*embs* vector) must be stored. This is not sufficient to store time constraints information.

Trail-based embedding database We use reversible vectors to store the start and the end positions of all the possible time-constrained embeddings for every sequence. The vectors: *sids*, emb_{size} and *embs* respectively represent the sequence ids, the number of embeddings and the start/end positions of the embeddings. These start and end positions are sufficient to verify the span and gap constraints during pattern extension.

Example 12 Figure 1b depicts an example of this datastructure: for pattern $\langle A\rangle$, the data of $embSDB|^{[3,7]}_{\langle A\rangle}$ is stored between indices $\phi = 1$ and $\phi+\varphi-1 = 4$. This pattern $\langle A\rangle$ is then further extended by the symbol D and $embSDB|^{[3,7]}_{\langle AD\rangle}$ is stacked next to it, between indices $\phi = 5$ and $\phi + \varphi - 1 = 7$. This pattern $\langle AD\rangle$ is then further extended by the symbol C and $embSDB|^{[3,7]}_{\langle ADC\rangle}$ is stacked between indices $\phi = 8$ and $\phi+\varphi-1 = 10$. The $gap^{[3,7]}$-embedding of $\langle ADC\rangle$ in sid_2 is $(3, 4, 5)$ but we only store the start (3) and end (5) as $(3, 5)$ we can compute the valid extension window based on gap and span constraint.

6 *PPICt* global constraint under time constraints

This section presents *PPICt* (Prefix Projection Incremental Counting with time restrictions), our filtering algorithm for finding sequential patterns under gap and span

[1] Except some CP-Solvers such as Gecode, Oz/Mozart and Figaro.

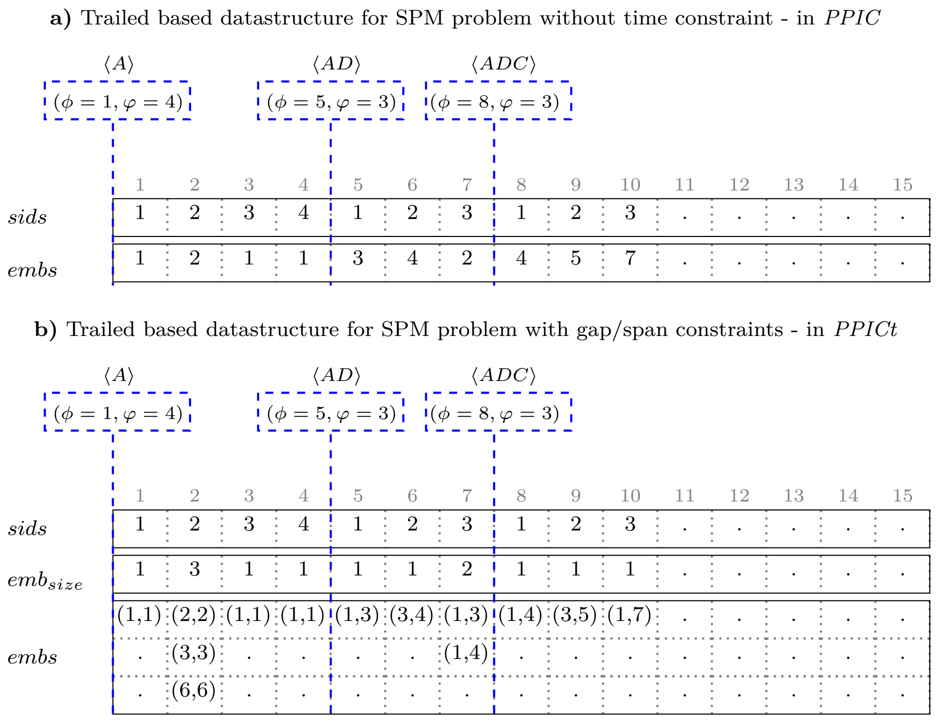

Fig. 1: Trailed-based datastructure to store and restore the embeddings database: **a)** without time constraints **b)** with $gap^{[3,7]}$ constraints

constraints. This algorithm support sequences with timestamps (as presented in Table 1a). Constraints such as regular expression, item inclusion/exclusion, pattern length and (theoretically) all other anti-monotonic constraints can be added to the model.

limitations The *PPICt* constraint does not support datasets with time-stamped sequences of itemset i.e. when a set of events without linear order occur at the same time. It also does not support multivariate time-series data [6] common in biological applications.

6.1 PPICt propagator

The PPICt$(P, SDB, \theta, M, N, W, Y)$ global constraint is given in Listing 1. It enforces the minimum frequency θ and the $gap^{[M,N]}$ and $span^{[W,Y]}$ constraints (1 to 5 from Definition 6) in a single propagator. The filtering procedure is triggered whenever a pattern is extended by a new symbol (P_{i+1}). If that symbol is ϵ then by Condition 2 all $P_j, j > i$ should also be ϵ. Then the non-prefix anti-monotone minimum span constraint must be checked in post-processing. If the pattern is still frequent then this is a new solution.

If P_{i+1} was assigned a *non-*ϵ value, then the procedure ProjectsAndGetFreqs counts for each symbol what the size of the projected database would be if the new pattern is extended with that symbol. The computed result, denoted $freqs$ in the pseudo-code, is used to prune the domain of the next pattern variable P_{i+1} by removing infrequent symbols; this is valid because the constraints filtered in ProjectsAndGetFreqs are *prefix anti-monotone*.

The main difference with *PPIC* [4] is that all embeddings must be stored instead of just the prefix (see Section 4). Embeddings can only be extended by symbols appearing in its *extension windows*. The projected frequency counting should only count symbols appearing in an extension window. Indeed, a symbol not appearing in an extension window of any embedding of the sequence would not be a valid support for extending the current pattern as it wouldn't satisfy the time constraints.

Listing 1: PPICt($P, SDB, \theta, M, N, W, Y$)

```
// pre: variables $\langle P_1, \ldots, P_i \rangle$ are bound, $SDB$ is given
//      $P_i$ is the new instantiated variable since previous call.
if ($P_i == \epsilon$)
  foreach (j $\in \{i+1, \ldots, L\}$)
    Remove all embeddings that do not satisfy minimum span $W$ and fail
        should the pattern no longer be frequent
    $P_j.assign(\epsilon)$ // Condition.2
else
  $freqs$ = ProjectAndGetFreqs($i, SDB, P_i, M, N, W, Y$)
  foreach ($a \in D(P_{i+1})$) if ($a \neq 0$ and $freqs[a] < \theta$) $P_{i+1}.remove(a)$
```

This leads to the following key ingredients of the ProjectsAndGetFreqs function: 1) as presented in Section 5, we adapt the backtracking-aware datastructure introduced in [4] to store all possible occurrences of the pattern in a sequence, including the starting symbol to support *span* constraints; 2) we avoid scanning a sequence for a symbol beyond the (precomputed) last occurrence of that symbol in the sequence; 3) we introduce the concept of *extension window* of an embedding and avoid to scan overlapping windows multiple times; 4) we avoid searching for the position of the start of an extension window, which depends on the minimum gap time, by precomputing these position in advance. These ingredients are detailed next.

6.1.1 Ingredient 1. Avoid scanning all sequences

We reuse the $lastPosMap$ precomputed structure of *PPIC* to avoid scanning a sequence if the last position of that sequence is before the start of the extension window. For a symbol a, the $lastPosMap[a]$ is the last position of this symbol in the sequence: $lastPosMap[a] = \max\{p \leq size(s) : s[p] = a\}$.

Example 13 Assuming the $lastPosMap$ precomputed structure provided in Table 1c and the symbol A, $lastPosMap[A]$ is $\{1, 6, 1, 1\}$. Hence, when searching for A, we must stop at the first position for the sequences sid_1, sid_3 and sid_4 but for the sequence sid_2 we stop at the position 6.

However, we cannot use the same structure for support counting (which also need to search symbols over sequences) as *PPIC* did, since this assumes that all symbols up to the end of the sequence must be counted, while we should only count symbols in the extension windows. This can have a big impact if the sequences contain many duplicates symbols as shown in [4]. In our case, it does not matter since extension windows are often smaller.

6.1.2 Ingredient 2. Avoid scanning more than once the events occurring in overlapping extension windows

The extension windows of a sequence can possibly overlap. For instance in Table 2b with $\alpha = \langle AD \rangle$, in sid_3, $\langle (E, 12) \rangle$ is present in both extension windows. Then when computing the $freqs$ vector, some positions could be revisited several times. This source of inefficiency can be avoided by keeping track of the current largest position visited so far in any extension window. This position is denoted pos in Listings 2 and 3. When the next extension window for the current sequence is considered by updateSupport in Listing 3, all symbols before pos have already been counted, so only positions after pos should be visited and afterwards pos is updated.

6.1.3 Ingredient 3. Avoid scanning the sequences for minimum gap constraint

Given the current pattern $\alpha = \langle \alpha_1 \alpha_2 \ldots \alpha_k \rangle$, a sequence s and a valid $gap^{[M,N]}span^{[W,Y]}$-embedding $e = (e_1, e_2, \ldots, e_k)$, all the symbols in the extension window $\text{ew}_e^{gap^{[M,N]}span^{[W,Y]}}(s)$ must be visited for updating the frequency counters. While it is easy to compute the start *time* of the extension window using the minimum gap M: $t_{e_k} + M$; finding the first position u in the sequence such that $t_u \geq t_{e_k} + M$ requires scanning the sequence starting from e_k. To avoid this, we propose to precompute, for the given minimum gap, the position of the beginning of the extension window from any possible position. This can be done with one linear scan over each sequence; and the precomputed positions are stored in a structure called *nextPosGap*.

Definition 9 Building the nextPosGap structure. Assume $s = \langle (s_1, t_1)(s_2, t_2) \ldots (s_n, t_n) \rangle$ is a sequence. Given $k \in [1, n]$ a position in s and M a minimum gap, the $nextPosGap[s][k]$ is the position of the *smallest time* satisfying the minimum gap: $nextPosGap[s][k] = i$ such that $(i > k) \wedge (t_i \geq t_k + M) \wedge (\nexists i' < i : t_{i'} \geq t_k + M)$.

Example 14 Assume $s = \langle (A, 2)(B, 5), (D, 6), (C, 10)(B, 11) \rangle$, $k = 2$ and $M = 3$ $nextPosGap[s][k] = 4$ because $t_4 = 10$ is the smallest time such that $t_4 \geq 5 + 3 = 8$. Table 1b shows the $nextPosGap$ of SDB (the values $nextPosGap[s][k] > size(s) + 1$ means the minimum gap is not available for that position) .

6.1.4 Putting it all together

The core of the algorithm is in the ProjectAndGetFreqs procedure (presented in Listing 2) that gathers all the ingredients. We distinguish two cases. If $l == 1$ it means that the pattern was previously empty and is now composed of one unique symbol. If $(l > 1)$ the pattern is composed of at least two symbols which means that the gap/span must be considered.

In the first case $(l == 1)$, all sequences of SDB are considered. For every sequence, all the positions having the symbol a are stored as an embedding. As the embedding is a singleton, there is not need to consider the gap/span constraints at this point.

Listing 2: ProjectAndGetFreqs(l, SDB, a, M, N, W, Y)

```
 1 // Internal state: $\phi, \varphi, sids, emb_{size}, embs$
 2 $\phi' = \phi + \varphi$; $\varphi' = 0$; $freqs[b] = 0\ \ \forall b \in \{0, \ldots, L\}$
 3 if $l == 1$: // first assigned symbol, scan for symbol
 4   for $sid = 1$ to $SDB.size$: // for every sequence in SDB
 5     $seq$ = SDB[$sid$]; $nEmb = 0$; $pos = 0$; $visitedI[b] = false\,\forall b \in \{0, \ldots, L\}$
 6     for $i = 0$ to $lastPosMap[sid][a]$: // find each symbol $a$
 7       if $seq[i] == a$: // new match
 8         $embs[\phi' + \varphi'][nEmb] = (i, i)$; $nEmb = nEmb + 1$
 9         $pos = \text{updateSupport}(i, i, sid, pos, visitedI)$
10         if ($pos \geq seq.size$) break // window ends with sequence
11     if ($nEmb > 0$)// store sequence meta-data
12       $sids[\phi' + \varphi'] = sid$; $emb_{size}[\phi' + \varphi'] = nEmb$; $\varphi' = \varphi' + 1$
13 else: // non-empty prefix
14   for $c = \phi$ to $\phi + \varphi - 1$: //for all sequence in projected database
15     $sid = sids[c]$; seq = SDB[$sid$]; $nEmb = 0$; $pos = 0$
16     $visitedI[b] = false\,\forall b \in \{0, \ldots, H\}$
17     for $k = 1$ to $emb_{size}[c]$: // for each prefix embedding
18       $(b, e) = embs[c][k]$ // begin and end position of embedding
19       $maxT = min(seq^t[sid][b] + Y, seq^t[sid][e] + N)$ // max time window
20       $i = nextPosGap[sid][e]$ // precomputed position of minT
21       while ($i < lastPosMap[sid][a]$ and $seq^t[sid][i] \leq maxT$):
22         if $seq[i] == a$: // new embedding
23           $embs[\phi' + \varphi'][nEmb] = (b, i)$; $nEmb = nEmb + 1$
24           $pos = \text{updateSupport}(b, i, sid, pos, visitedI)$
25           if ($pos \geq seq.size$) break // window ends with sequence
26         $i = i + 1$
27   if ($nEmb > 0$) // store sequence meta-data
28     $sids[\phi' + \varphi'] = sid$; $emb_{size}[\phi' + \varphi'] = nEmb$; $\varphi' = \varphi' + 1$
29 $\phi = \phi'$; $\varphi = \varphi'$
30 return $freqs$
```

Listing 3: updateSupport$(b, e, sid, pos, visitedI)$

```
$s =$SDB$[sid]$; $k = max(nextPosGap[sid][e], pos)$
$maxT = min(s^t[sid][e] + N, s^t[sid][b] + Y)$
while ( $k < s.size$ and $s^t[sid][k] \leq maxT$ )
  if (!visitedI[$s[k]$] ) $freqs[s[k]] = freqs[s[k]] + 1$; visitedI[$s[k]$]$= true$
  $k = k + 1$
return k
```

The call to updateSupport (Listing 3) will update the $freqs$ vector by visiting each symbol present in the extension window of the current embedding (position of symbol a). Variable pos is used to avoid incrementing the frequency of a symbol twice in the same sequence.

In the second case ($l > 1$), the main loop at line 14 iterates over the previous *(parent)* projected database stored between ϕ and $\phi + \varphi - 1$ and builds the new one starting at index $\phi + \varphi$. For each embedding of a sequence, s (line 17), the maximum of the time window is computed. We search all positions only in extension window ensuring to have time greater than minimum gap time and lower than the maximum gap and span times computed based on the first and the last element of the embedding (line 19). The updateSupport is also called to update the $freqs$ vector for every extended embedding created.

Finally, lines 12 and 28 update *sids* and emb_{size} in order to ensure the consistency of the datastructure. Then, line 29 updates the reversible integers ϕ and φ to reflect the newly computed projected database.

6.1.5 Additional constraints

The advantage of CP based sequence mining is its capacity to accept additional constraints. The global constraint approach *PPICt* is less flexible than the decomposition approach of [25] as it does not expose the embedding variable. Nevertheless many useful string constraints [15] can be added on the sequence pattern variables: $P = [P_1, P_2 \ldots, P_l]$.

Pattern length constraints. One can impose a minimum and a maximum over the length of the pattern. These constraints are easy to handle considering all patterns are terminated by the empty symbol (ϵ). Hence, the minimum pattern length ($Lmin$) is defined as $\forall i \in [1, Lmin]$ and $Lmin < l$, $P_i \neq \epsilon$. The Maximum pattern length ($Lmax$) is obtained by limiting the length of P to $Lmax$: $P = [P_1, P_2 \ldots, P_{Lmax}]$ with $Lmax < l$.

Symbol inclusion/exclusion. The number of occurrences of symbols in the sequence pattern can be modeled with *Among* [7] and global cardinality [**?**] constraints largely available in CP solvers.

Regular expression. The *Regular* constraint [30] can be used to enforce that P satisfies a given regular expression. Most of CP-based approaches [4, 19, 20] support regular expression constraints.

6.2 Time and space complexity

Let us denote by $m = size(SDB)$ the number of sequences, l the length of the longest sequence, $L = |\mathcal{I}|$ the size of item alphabet, d the maximum depth of the tree search. In the worst case, the time complexity of our propagator is in $O(m \times l^2 + L)$ and the space complexity is in $O(m \times d \times l)$.

Proof (i) Space complexity. Our datatructure needs $O(l)$ memory entries to store the embeddings for one sequence. One projected database requires thus $O(m \times l)$. Down a branch of the search tree the space complexity is $O(m \times d \times l)$. *(ii) Time complexity. PPICt* needs $O(l + (m \times l^2 + L)) = O(m \times l^2 + L)$ time to be complete since the lines 4-6 of the Listing 1 cost $O(l)$, the ProjectAndGetFreqs method $O(m \times l^2)$ and the line 9 $O(L)$. The complexity of the loops 6 and 21 including the updateSupport in Listing 2 is $O(l)$ since we avoid scanning overlap. Hence, the complexity of ProjectAndGetFreqs method is $O(m \times l + m \times l \times l) = O(m \times l^2)$. □

7 Experiments

This section reports the experiments we made to evaluate our approach in comparison with other CP-based and specialized methods. More specially, we answer the following questions: *Q1.* What is the performance of the state-of-the-art for sequential pattern mining without time constraint? *Q2.* What is the difference in performance of *PPICt* for sequential patterns mining with time restrictions? *Q3.* What is the

SDB	$size(SDB)$	L	$avg(size(s))$	$avg(size(I_{/s}))$	$max(size(s))$	description
BIBLE	36369	13905	21.64	17.85	100	text
FIFA	20450	2990	36.24	34.74	100	web click stream
Kosarak	69999	21144	7.98	7.98	796	web click stream
Leviathan	5834	9025	33.81	26.34	100	text
Msnbc	31,790	17				web click stream
PubMed	17237	19931	29.56	24.82	198	bio-medical text
protein	103120	25	482.25	19.93	600	protein sequences

Table 3: Seven real-life datasets features. (*avg* stands for average)

effect of a standalone constraint in the mining process? *Q4.* What is the impact of the computation of the additional embeddings in *PPICt*?

Before answer these questions, we present in Table 3 the features of the seven real-life datasets that we use but also the experimental protocol and the alternative sequential pattern miners used for the comparisons. Notice data and the framework are available and open source[2].

7.1 Experimental protocol

PPICt is implemented in the Scala language in the CP-Solver OscaR [26]. All experiments are run in the JVM with maximum memory set to 8GB. All the experiments are conducted using a 2.7GHz Intel Core i5 64 bit processor and 8GB of RAM with Linux 3.19.0-32-generic from Mint 17.3. We set the execution time limit to 3600 seconds (1 hour). We also restrict the output of all software to only the mining statistics and do not print the patterns found. The minimum support θ is denoted by $Minsup$.

7.2 Alternative sequential patterns miners

We make comparisons with *GapSeq*[3] [19], a constraint programming approach that outperforms other CP-based methods supporting gap constraints; *cSPADE*[4] [38] a highly scalable specialized sequence miner that supports gap and span constraints. Its search is not based on pattern extension as *GapSeq* and *PPICt* are, but on repeated (temporal) joins of embeddings. We also provide a comparison to *PPIC*[5] [4] without gap constraints, *PPIC* has shown to outperform both specialized and generic miners for standard frequent sequence mining. Table 4 shows the supported constraints for these miners.

7.3 Performances results

7.3.1 Q1: GapSeq vs PPIC vs cSPADE for SPM without time restriction

As shown in [4] and illustrated in Fig. 2, *PPIC* clearly outperforms both CP-based and specialized approaches for many datasets (with several different features) except

[2] http://sites.uclouvain.be/cp4dm/spm/ppict/
[3] https://sites.google.com/site/cp4spm/
[4] http://www.cs.rpi.edu/~zaki/www-new/pmwiki.php/Software
[5] http://sites.uclouvain.be/cp4dm/spm/

Methods	Frequency	Gap	Span	Regular	Among	Length	other constraints[1]
PPICt	x	x	x	x	x	x	x
GapSeq	x	x*			x	x	x
cSPADE	x	x	x**			x	

Table 4: Sequential patterns miners with supported constraints. [1]It is other anti-monotone constraints (not implmented but could be). * *GapSeq* doesn't consider time but position of events, ** *cSPADE* doesn't support minimum span constraint.

for the densest dataset Kosarak-70k where it is competitive with *cSPADE*. For very sparse dataset like protein *PPIC* is at least one hundred times faster.

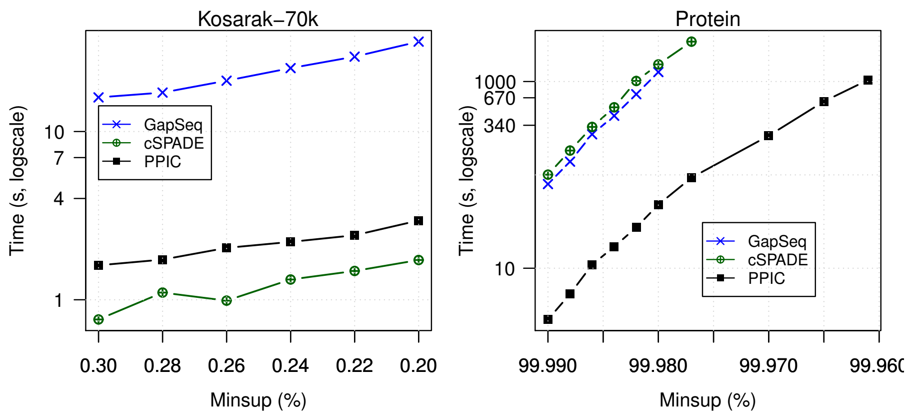

Fig. 2: CPU times for PPIC (without time constraints) with several minsup (missing points indicate a timeout) [4]

7.3.2 Q2: Time performance for PPICt under gap and span constraints

We first compare *PPICt* with *GapSeq* and *PPIC* for gap constraints. Then, we combine gap and span constraints.

Figure 3 shows the CPU time for the sequence mining task under minimum and maximum gap for several θ ($Minsup$) values over six datasets. *PPICt* clearly outperforms both CP-based and specialized methods. Except for the Kosarak dataset, PPICt is always faster, and increasingly so for low frequency thresholds. Upon inspecting the output of the Kosarak dataset we see that several frequent patterns have the same size and cover the same set of sequences. The temporal join approach used by *cSPADE* is very fast in this case. This was also the case for *PPIC* in the non-time constrained case.

We also combine the *gap* and *span* constraints, which is not supported by GapSeq. The results are presented in Fig. 4. Our approach outperforms *cSPADE* by a wide

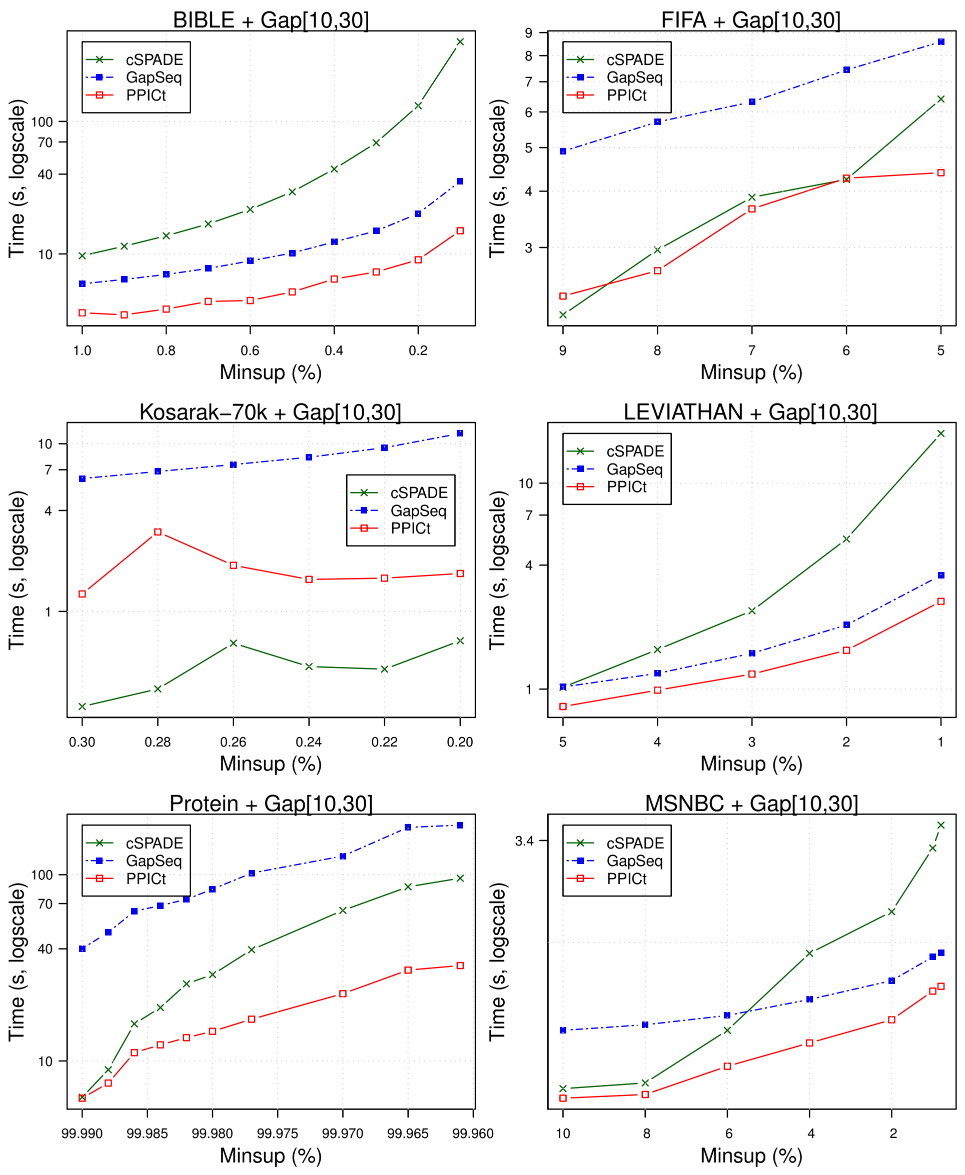

Fig. 3: CPU times when considering minimum and maximum gap constraints for several minsup (missing points indicate a timeout)

margin in this case. These results show that *PPICt* is still efficient when combining time constraints.

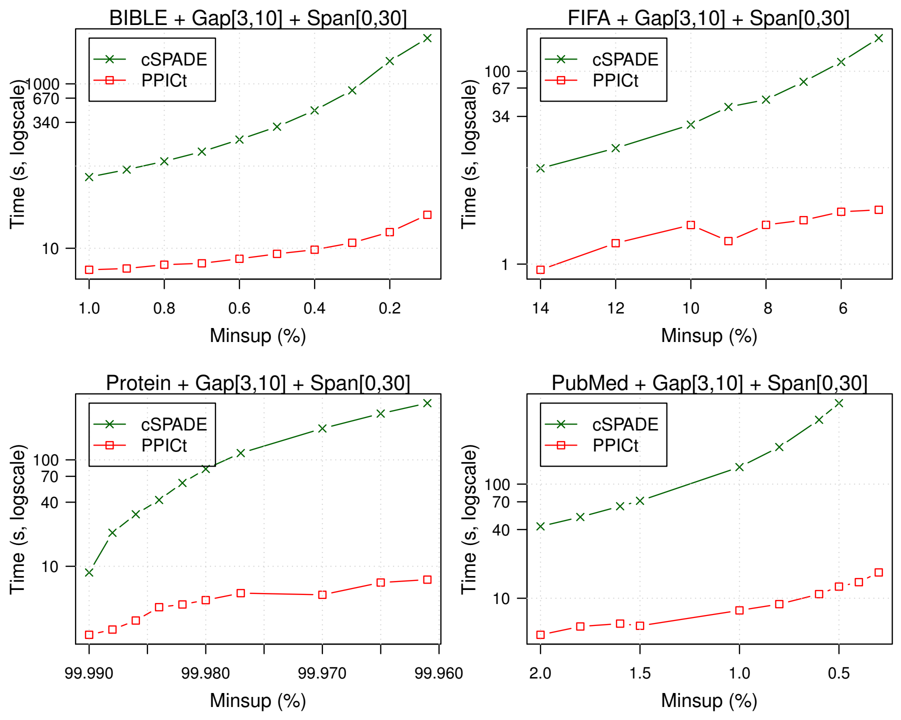

Fig. 4: CPU times when considering both *gap* and *span* constraints for several minsup (missing points indicate a timeout).

7.3.3 Q3: Effect of maximum gap constraint

We now look at the sensitivity of the methods to the threshold of the maximum gap constraint. We fix the frequency threshold to a low value that makes mining without further constraints challenging and increase the maximum gap constraint from 1 to 9. As can be seen in Fig. 5, the runtime of *cSPADE* increases much more quickly with increasing maximum gap. For *GapSeq* it depends on the dataset, but *PPICt*'s performance is more stable and increases more moderately compared to the other methods.

7.3.4 Q4: Experiments over databases without time restrictions

To answer *Q4*., we use *PPICt* to find only the sequential patterns without any time considerations. That is *PPICt* where minimum gap/span is 0 and maximum gap/span is the infinity, denoted by *PPICt[0,Inf]*. Hence, we compare *PPIC* with *PPICt[0,Inf]*. The results are reported in Fig. 6. We can notice that *PPIC* is always faster. This is possible since such *PPIC* improvements could not be used under time restrictions. Moreover, to preserve the structure of datasets the reduction of datasets by preprocessing is forbidden.

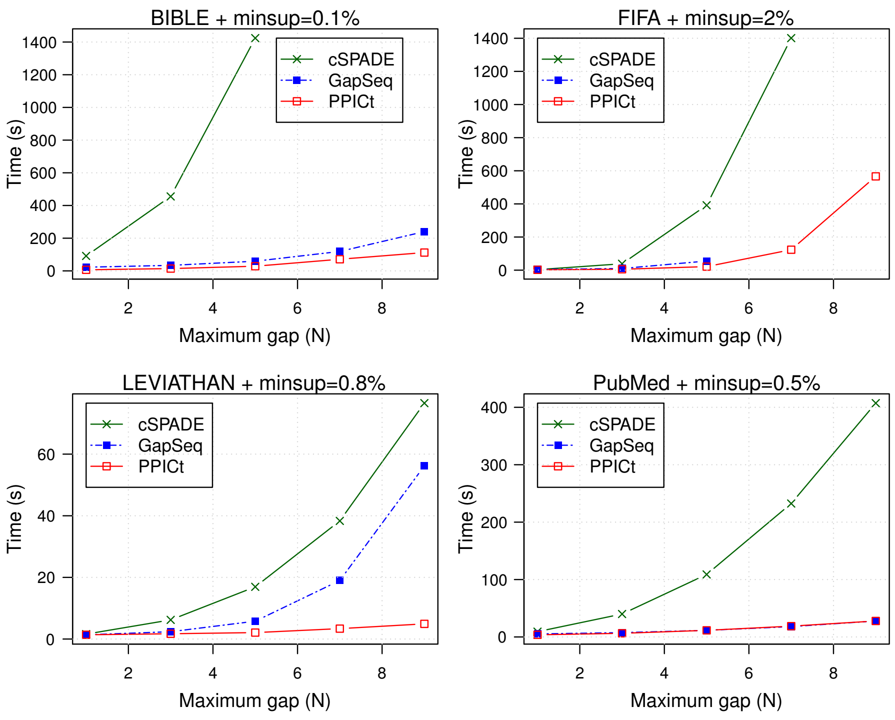

Fig. 5: CPU times for several maximum gap with fixed minsup over Bible, Fifa, Leviathan and PubMed datasets (missing points indicate a timeout).

	Gap		+Pattern Length		+Among		+Regular	
	nSols	time(s)	nSols	time(s)	nSols	time(s)	nSols	time(s)
BIBLE[6]	32307	46.181	1542	45.622	171	43.390	8	0.191
PubMed[7]	13086	22.632	1304	21.600	235	19.889	3	0.091

Table 5: Combination of pattern length, item inclusion/exclusion, regular expression constraints with gap constraint.

7.4 Handling additional Constraints

To demonstrate the ability to accommodate additional constraints we experiment the combination of *PPICt* with some other sequence pattern constraints. The result is shown in Table 5. We can observe that the addition of the constraints reduces the number of solutions and the computation time while a *generate-and-filter* approach using a dedicated algorithm would not benefit from a stronger filtering.

[6] $\theta = 0.1\% + Gap[10,30] + (Lmin = Lmax = 5)$+the number A equal $1+E$ is forbidden+Regular$(A+(B\{2,\}|C*|D+)B*C*D*)$ where ($A = 11829$, $B = 2$, $C = 8212$, $D = 6556$, $E = 5590$)

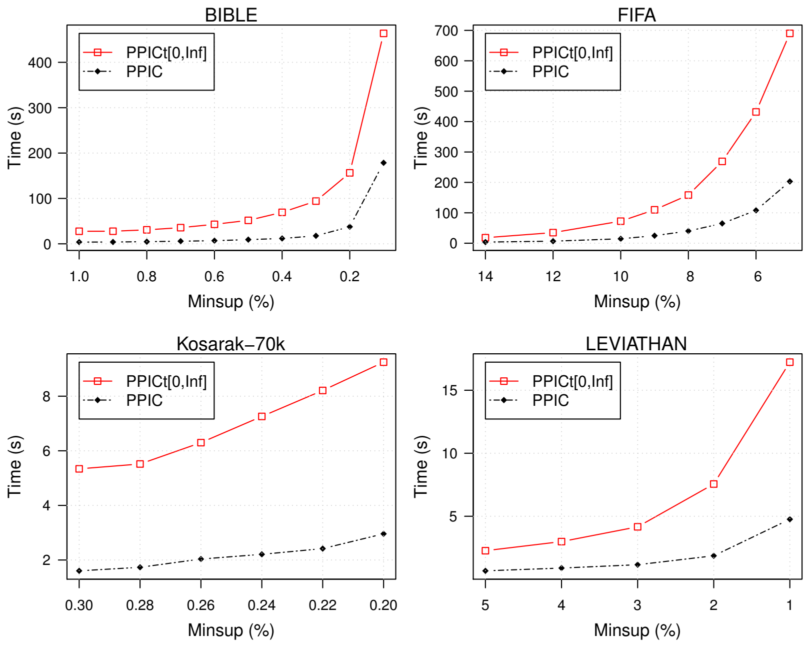

Fig. 6: Comparing *PPICt* without time restriction (*PPICt[0,Inf]*) with *PPIC*

8 Conclusion

We introduced *PPICt*, a global constraint to solve sequential pattern mining problem under time constraints. It integrates *gap* and *span* constraints for database with or without timestamps. Our approach often outperforms *cSPADE*, the state-of-the-art specialized method and always outperforms *GapSeq*, the state-of-the-art CP based approach allowing to handle time constraints. This was made possible thanks to the backtracking-aware datastructure to store embeddings of pattern based on trailing techniques. Also, algorithmic ingredients help to improve more: the precomputed next position of minimum gap, the avoidance of scanning all dataset and the avoidance of the overlapping between extension windows when computing the frequencies of symbols. Moreover, we report experimental results over several real-life datasets which demonstrate that our proposal is mostly competitive with or outperforms both specialized and CP-based methods. Additional constraints such as regular expression, item inclusion/exclusion, pattern length constraints are also available to increase the flexibility of users and practitioners.

[7] $\theta = 0.3\%$+$Gap[10, 30]$+$(Lmin = Lmax = 4)$+the number A and B equal 1+Regular$(B + A * C * A*)$ where ($A = 3335$, $B = 12155$, $C = 16599$)

Acknowledgments

The first author is supported by the FRIA-FNRS (Fonds pour la Formation à la Recherche dans l'Industrie et dans l'Agriculture, Belgium).

References

1. Aggarwal, C.C., Han, J.: Frequent pattern mining. Springer (2014)
2. Agrawal, R., Srikant, R.: Mining sequential patterns. In: Data Engineering, 1995. Proceedings of the Eleventh International Conference on, pp. 3–14. IEEE (1995)
3. Antunes, C., Oliveira, A.L.: Generalization of Pattern-Growth Methods for Sequential Pattern Mining with Gap Constraints, pp. 239–251. Springer (2003)
4. Aoga, J.O., Guns, T., Schaus, P.: An efficient algorithm for mining frequent sequence with constraint programming. LNAI,Part II, ECML PKDD **9853** (2016)
5. Ayres, J., Flannick, J., Gehrke, J., Yiu, T.: Sequential pattern mining using a bitmap representation. In: ACM SIGKDD, pp. 429–435 (2002)
6. Batal, I., Fradkin, D., Harrison, J., Moerchen, F., Hauskrecht, M.: Mining recent temporal patterns for event detection in multivariate time series data. In: Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 280–288. ACM (2012)
7. Beldiceanu, N., Contejean, E.: Introducing global constraints in chip. Mathematical and computer Modelling **20**(12), 97–123 (1994)
8. Coquery, E., Jabbour, S., Saïs, L., Salhi, Y.: A SAT-based approach for discovering frequent, closed and maximal patterns in a sequence. In: L.D. Raedt, C. Bessière, D. Dubois, P. Doherty, P. Frasconi, F. Heintz, P.J.F. Lucas (eds.) ECAI, *Frontiers in Artificial Intelligence and Applications*, vol. 242, pp. 258–263. IOS Press (2012). URL http://www.booksonline.iospress.nl/Content/View.aspx?piid=31572
9. Desai, N.A.K., Ganatra, A.: Efficient constraint-based sequential pattern mining (spm) algorithm to understand customers buying behaviour from time stamp-based sequence dataset. Cogent Engineering **2**(1), 1072,292 (2015)
10. Fournier-Viger, P., Wu, C.W., Tseng, V.S.: Mining maximal sequential patterns without candidate maintenance. In: Advanced Data Mining and Applications, pp. 169–180. Springer (2013)
11. Golden, K., Pang, W.: Constraint Reasoning over Strings, pp. 377–391. Springer Berlin Heidelberg, Berlin, Heidelberg (2003). DOI 10.1007/978-3-540-45193-8_26. URL http://dx.doi.org/10.1007/978-3-540-45193-8_26
12. Guns, T., Nijssen, S., De Raedt, L.: k-pattern set mining under constraints. Knowledge and Data Engineering, IEEE Transactions on **25**(2), 402–418 (2013)
13. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In: ACM Sigmod Record, vol. 29, pp. 1–12. ACM (2000)
14. Han, J., Pei, J., Yin, Y., Mao, R.: Mining frequent patterns without candidate generation: A frequent-pattern tree approach. Data mining and knowledge discovery **8**(1), 53–87 (2004)
15. He, J., Flener, P., Pearson, J., Zhang, W.M.: Solving string constraints: The case for constraint programming. In: International Conference on Principles and Practice of Constraint Programming, pp. 381–397. Springer (2013)
16. Henriques, R., Antunes, C., Madeira, S.C.: Methods for the Efficient Discovery of Large Item-Indexable Sequential Patterns, pp. 100–116. Springer International Publishing, Cham (2014). DOI 10.1007/978-3-319-08407-7_7. URL http://dx.doi.org/10.1007/978-3-319-08407-7_7
17. Henriques, R., Madeira, S.C.: Bicspam: flexible biclustering using sequential patterns. BMC Bioinformatics **15**(1), 130 (2014). DOI 10.1186/1471-2105-15-130. URL http://dx.doi.org/10.1186/1471-2105-15-130
18. Kemmar, A., Lebbah, Y., Loudni, S., Boizumault, P., Charnois, T.: Prefix-projection global constraint and top-k approach for sequential pattern mining. Constraints **22**(2), 265–306 (2017). DOI 10.1007/s10601-016-9252-z. URL http://dx.doi.org/10.1007/s10601-016-9252-z
19. Kemmar, A., Loudni, S., Lebbah, Y., Boizumault, P., Charnois, T.: A global constraint for mining sequential patterns with gap constraint. CPAIOR16 (2015)

20. Kemmar, A., Loudni, S., Lebbah, Y., Boizumault, P., Charnois, T.: Prefix-projection global constraint for sequential pattern mining. In: Principles and Practice of Constraint Programming. Springer (2015)
21. Li, C., Wang, J.: Efficiently mining closed subsequences with gap constraints. In: SDM, pp. 313–322. SIAM (2008)
22. Lu, S., Li, C.: Aprioriaadjust: An efficient algorithm for discovering maximal sequential patterns (2004)
23. Mannila, H., Toivonen, H., Verkamo, A.I.: Discovery of frequent episodes in event sequences. Data mining and knowledge discovery **1**(3), 259–289 (1997)
24. Metivier, J., Boizumault, P., Crémilleux, B., Khiari, M., Loudni, S.: A constraint-based language for declarative pattern discovery. In: Data Mining Workshops (ICDMW), 2011 IEEE 11th International Conference on, pp. 1112–1119. IEEE (2011)
25. Negrevergne, B., Guns, T.: Constraint-based sequence mining using constraint programming. In: CPAIOR15. Springer (2015)
26. OscaR Team: OscaR: Scala in OR (2012). Available from https://bitbucket.org/oscarlib/oscar
27. Parthasarathy, S., Zaki, M.J., Ogihara, M., Dwarkadas, S.: Incremental and interactive sequence mining. In: Proceedings of the eighth international conference on Information and knowledge management, pp. 251–258. ACM (1999)
28. Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U., Hsu, M.C.: Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In: icccn, p. 0215. IEEE (2001)
29. Pei, J., Han, J., Wang, W.: Constraint-based sequential pattern mining: the pattern-growth methods. Journal of Intelligent Information Systems **28**(2), 133–160 (2007). DOI 10.1007/s10844-006-0006-z. URL http://dx.doi.org/10.1007/s10844-006-0006-z
30. 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)
31. Pinto, H., Han, J., Pei, J., Wang, K., Chen, Q., Dayal, U.: Multi-dimensional sequential pattern mining. In: Proceedings of the tenth international conference on Information and knowledge management, pp. 81–88. ACM (2001)
32. Quimper, C.G., Walsh, T.: Global grammar constraints. In: International Conference on Principles and Practice of Constraint Programming, pp. 751–755. Springer (2006)
33. Rossi, F., Van Beek, P., Walsh, T.: Handbook of CP. Elsevier (2006)
34. Srikant, R., Agrawal, R.: Mining sequential patterns: Generalizations and performance improvements. Springer (1996)
35. Wang, J., Han, J., Li, C.: Frequent closed sequence mining without candidate maintenance. Knowledge and Data Engineering, IEEE Transactions on **19**(8), 1042–1056 (2007)
36. Yan, X., Han, J., Afshar, R.: Clospan: Mining closed sequential patterns in large datasets. In: In SDM, pp. 166–177. SIAM (2003)
37. Zaki, M.J.: Efficient enumeration of frequent sequences. In: Proceedings of the seventh international conference on Information and knowledge management, pp. 68–75. ACM (1998)
38. Zaki, M.J.: Sequence mining in categorical domains: incorporating constraints. In: Proceedings of the ninth international conference on Information and knowledge management, pp. 422–429. ACM (2000)
39. Zaki, M.J.: Spade: An efficient algorithm for mining frequent sequences. Machine learning **42**(1-2), 31–60 (2001)
40. Zhao, Q., Bhowmick, S.S.: Sequential pattern mining: A survey. ITechnical Report CAIS Nayang Technological University Singapore pp. 1–26 (2003)