# CCSM: an efficent algorithm for constrained sequence mining

S. Orlando[1], R. Perego[2], C. Silvestri[1]

[1]Dipartimento di Informatica, Università Ca' Foscari, Venezia, Italy
[2]Istituto ISTI, Consiglio Nazionale delle Ricerche (CNR), Pisa, Italy
{orlando, claudio}@dsi.unive.it, raffaele.perego@cnuce.cnr.it

## Abstract

This paper proposes CCSM (Cache-based Constrained Sequence Miner), a new level-wise algorithm that mines temporal databases to find sequential patterns satisfying user-defined constraints. The main innovation of CCSM is the adoption of $k$-way intersections of idlists to compute the support of candidate sequences. Our $k$-way intersection method is enhanced by the use of an effective *cache* that stores intermediate idlists for future reuse. The exploitation of the cache entails a surprising reduction in the actual number of join operations performed on idlists. Moreover, CCSM is able to deal with very complex constraints, like the maximum temporal gap between events occurring in the input sequences.

We experimentally evaluated the performances of CCSM on synthetically generated datasets, and compared them with those obtained running the cSPADE algorithm on the same datasets.

## 1 Introduction

The problem of mining frequent sequential patterns was introduced by Agraval and Srikant in [2]. In a subsequent work the same authors discussed the introduction of constraints on the mined sequences, and proposed GSP [9], a new algorithm dealing with them. In the last years, many innovative algorithms were presented for solving the same problem, also under different user-provided constraints [10, 11, 7, 4, 6, 3].

We can think of the problem of mining Frequent Sequential Patterns (FSP) as a generalization of Frequent Pattern (FP) mining to temporal databases. FP mining aims to find patterns (itemsets) occurring with a given minimum support within a transactional database $\mathcal{D}$, whose transactions correspond to collections of items. A pattern is frequent if its support is greater than (or equal to) a given threshold $s\%$, i.e. if it is set-included in at least $s\% \cdot |\mathcal{D}|$ input transactions, where $|\mathcal{D}|$ is the total number of transactions of $\mathcal{D}$. An input database $\mathcal{D}$ for the FSP problem is instead composed of a collection of *sequences*. Each sequence corresponds to a temporally ordered list of events, where each event is a collection of items (itemset) occurring simultaneously. The temporal ordering among the events is induced from the absolute timestamps associated with the events.

A sequential pattern is frequent if its support is greater than (or equal to) a given threshold $s\%$, i.e. if it is "contained" in (or it is a *subsequences* of) at least $s\% \cdot |\mathcal{D}|$ input sequences, where $|\mathcal{D}|$ is the number of sequences included in $\mathcal{D}$.

To make more intuitive both problem formulations, we may consider them within the applicative context of the market basket analysis (MBA). In this context, each transaction (itemset) occurring in a database $\mathcal{D}$ of the FP problem corresponds to the collection of items purchased by a customer during a single visit to the market. FP mining for MBA consists in finding frequent associations among the items purchased by customers. In the general case, we are thus neither interested in the timestamp of each purchased basket, nor in the identity of its customer, so the input database does not need to store such information. Conversely, FSP mining for MBA consists in predicting customer behaviors on the basis of their past purchases. Thus, $\mathcal{D}$ has also to include information about timestamp and customer identity of each basket. The sequences of events included in $\mathcal{D}$ correspond to sequences of "baskets" (transactions) purchased by the same customer during distinct visits to the market, and the items of a sequential pattern can span a set of subsequent transactions belonging to the same customer. Thus, while the FP problem is interested in finding intra-transaction patterns, the FSP problem determines inter-transaction sequential patterns.

Due to the similarities between the FP e FSP problems, several FP algorithms have been adapted for mining frequent sequential patters as well. Like FP algorithms, also FSP ones can adopt either a *count-based* or *intersection-based* approach for determining the support of frequent patterns. The GSP algorithm, which is derived from Apriori [1], adopts a count-based approach, together with a level-wise visit (Breadth-First) of the search space. At each iteration $k$, a set of candidate $k$-sequences (sequences of length $k$) is generated, and the dataset, stored in horizontal form, is scanned to count how many times each candidate is contained within each input sequences. The other approach, i.e. the intersection-based one, relies on a vertical-layout database, where for each item $X$ appearing in the various input sequences we store an *idlist* $\mathrm{L}(X)$. The idlist contains information about the identifiers of the input sequences (*sid*) that include $X$, and the timestamps (*eid*) associated with each occurrence of $X$. Idlists are thus composed of pairs (*sid*, *eid*), and are considerably more complex than the lists of transaction identifiers (*tidlists*) exploited by intersection-based FP algorithms. Using an intersection-based method, the support of a candidate is determined by joining lists. In the FP case, tidlist joining is done by means of simple set-intersection operations. Conversely, idlist joining in FSP intersection-based algorithms exploits a more complex temporal join operation. The Zaki's SPADE algorithm [11] is the best representative of such intersection-based FSP algorithms.

Several real applications of FSP mining enforces specific *constraints* on the type of sequences extracted [9, 7]. For example, we might be interested in finding frequent sequences of purchase events which contain a given subsequence (super pattern constraint), or where the average price of items purchased is over a given threshold (aggregate constraint), or where the temporal intervals between each pair of consecutive purchases is below a given threshold (max_gap constraint). Obviously we could solve this problem with a post-processing phase: first we extract from the database all the frequent sequences, and then we filter them on the basis of the posed constraints. Unfortunately, when the constraint is not on the sequence itself but on its occurrences (as in the case of of the max_gap constraint), sequence filtering requires an additional scan of the database to verify whether a given frequent pattern has still a minimum support under the constraint. In general, FSP algorithms that directly deal with user-provided constraints during the mining process are much more efficient, since constraints may involve an effective prune of candidates, thus resulting in a strong reduction of the computational cost. Unfortunately, the inclusion of some support-related constraints may require large modifications in the code of an unconstrained FSP algorithm. For example, the introduction of the max_gap constraint in the SPADE algorithm, gave rise to cSPADE, a very different algorithm [10].

All the FSP algorithms rely on the anti-monotonic property of sequence frequency: all the subsequences of a frequent sequence are frequent as well. This property is used by level-wise algorithms to generate candidate $k$-sequence from frequent $(k-1)$-sequences. When an intersection-based approach is adopted, we can determine the support of any $k$-sequence by means of join operations performed [8] on the idlist associated with its subsequences. As a limit case, we could compute the support of a sequence by joining the atomic idlists associated with the single items included in the sequence, i.e., through a $k$-way join operation [5]. More efficiently, we could compute the support of a sequence by joining the idlists associated with two generating $(k-1)$-subsequences, i.e., through a 2-way join operation. SPADE [11] just adopts this 2-way intersection method, and computes the support of a $k$-sequence by joining two of its $(k-1)$-subsequences that share a common suffix. Unfortunately, the adoption of 2-way intersections requires to maintain the idlists of all the $(k-1)$-subsequences computed during the previous iteration. To limit memory requirement, SPADE subdivides the search space into small, manageable chunks. This is accomplished by exploiting suffix-based equivalence classes: two $k$-sequences are in the same class only if they share a common $(k-1)$-suffix. Since all the generating subsequences of a given sequence belong to the same equivalence class, equivalence classes are used to partition the search space in a way that allow each class to be processed independently in memory. Unfortunately, the efficient method used by SPADE to generate candidates and join their idlists, cannot be exploited when a maximum gap constraint is considered. Therefore, cSPADE is forced to adopt a different and much more expensive way to generate sequences and join idlists, also maintaining in memory $\mathrm{F}_2$, the set of frequent 2-sequences.

In this paper we discuss CCSM (Cache-based Constrained Sequence Miner), a new level-wise intersection-based FSP algorithm, dealing with the challenging maximum gap constraint. Similarly to other sequence mining algorithms, also our intersection-based algorithm was inspired by a previously proposed FP algorithm (DCI [5]). The main

innovation of CCSM, inherited from DCI, is the adoption of $k$-way intersections to compute the support of candidate sequences. Our $k$-way intersection method is enhanced by the use of an effective *cache* which store intermediate idlists. The *idlist reuse* allowed by our cache entails a surprising reduction in the actual number of join operations performed, so that the number of joins performed by CCSM approaches the number of joins performed when a pure 2-way intersection method is adopted, but require much less memory. In this context, it becomes interesting to compare the performances of CCSM with the ones achieved by cSPADE when a maximum gap constraint is enforced.

The rest of the paper is organized as follows. Section 2 formally defines the FSP problem, while Section 3 describes the CCSM algorithm. In Section 4 we report and discuss experimental results. Finally Section 5 presents some concluding remarks.

# 2 Sequential patterns mining

## 2.1 Problem statement

**Definition 1. (Sequence of events)** *Let* $\mathcal{I} = \{i_1, ..., i_m\}$ *be a set of* m *distinct* items. *An* event (itemset) *is a non-empty subset of* $\mathcal{I}$. *A* sequence *is a temporally ordered list of events. We denote an event as* $(j_1, \ldots, j_m)$ *and a sequence as* $(\alpha_1 \rightarrow \ldots \rightarrow \alpha_k)$, *where each* $j_i$ *is an item and each* $\alpha_i$ *is an event* ($j_i \in \mathcal{I}$ *and* $\alpha_i \subseteq \mathcal{I}$). *The symbol* $\rightarrow$ *denotes a* happens-after *relationship. The items that appear together in an event* happen simultaneously. *The* length $|x|$ *of a sequence* $x$ *is the number of items contained in the sequence* ($|x| = \sum |\alpha_i|$). *A sequence of length* $k$ *is called a* $k$*-sequence.*

Even if an event represents a set of items occurring simultaneously, it is convenient to assume that there exists an ordering relationship $R$ among them. Such order makes univocal the way in which a sequence is written, e.g., we can not write $BA \rightarrow DBF$ since the correct way is $AB \rightarrow BDF$. This allows us to say, without ambiguity, that the sequence $A \rightarrow BD$ is a prefix of $A \rightarrow BDF \rightarrow A$, while $DF \rightarrow A$ is a suffix. A prefix/suffix of a given sequence $\alpha$ are particular subsequences of $\alpha$ (see Def. below).

**Definition 2. (Subsequence)** *A sequence* $\alpha = (\alpha_1 \rightarrow \ldots \rightarrow \alpha_k)$ *is* contained *in a sequence* $\beta = (\beta_1 \rightarrow ... \rightarrow \beta_m)$ *(denoted as* $\alpha \preceq \beta$*), if there exists integers* $1 \leq i_1 < ... < i_k \leq m$ *such that* $\alpha_1 \subseteq \beta_{i_1}, ..., \alpha_k \subseteq \beta_{i_k}$. *We also say that* $\alpha$ *is a* subsequence *of* $\beta$, *and that* $\beta$ *is a* super-sequence *of* $\alpha$.

**Definition 3. (Database)** *A temporal database is a collection of* input sequences*:*

$$\mathcal{D} = \{\overline{\alpha} |\ \overline{\alpha} = (sid, \alpha, eid)\},$$

*where* sid *is a sequence identifier,* $\alpha = (\alpha_1 \rightarrow \ldots \rightarrow \alpha_k)$ *is an event sequence, and* $eid = (eid_1, \ldots, eid_k)$ *is a tuple of unique* event identifiers*, where each* $eid_i$ *is the timestamp (occurring time) of event* $\alpha_i$.

**Definition 4. (Gap constrained occurrence of a sequence)** *Let* $\overline{\beta}$ *a given* input sequence, *whose events* $(\beta_1 \rightarrow \ldots \rightarrow \beta_m)$ *are timestamped with* $(eid_1, \ldots, eid_m)$. *The* gap *between two consecutive events* $\beta_i$ *and* $\beta_{i+1}$ *is thus defined as* $(eid_{i+1} - eid_i)$.

*A sequence* $\alpha = (\alpha_1 \rightarrow \ldots \rightarrow \alpha_k)$ occurs *in* $\overline{\beta}$ *under the* minimum gap *and* maximum gap *constraints, denoted as* $\alpha \sqsubseteq_c \overline{\beta}$, *if there exists integers* $1 \leq i_1 < ... < i_k \leq m$ *such that* $\alpha_1 \subseteq \beta_{i_1}, ..., \alpha_k \subseteq \beta_{i_k}$, *and* $\forall j,\ 1 < j \leq k$, $min\_gap \leq (eid_{i_j} - eid_{i_{j-1}}) \leq max\_gap$, *where* $min\_gap$ *and* $max\_gap$ *are user specified thresholds.*

*When no constraints are specified, we denote the occurrence of* $\alpha$ *in* $\overline{\beta}$ *as* $\alpha \sqsubseteq \overline{\beta}$. *This case is a simpler case of sequence* occurrence*, since we have that* $\alpha \sqsubseteq \overline{\beta}$ *simply if* $\alpha \preceq \beta$ *holds.*

**Definition 5. (Support and constraints)** *The* support *of a sequence pattern* $\alpha$, *denoted as* $\sigma(\alpha)$, *is the number of distinct input sequences* $\overline{\beta}$ *such that* $\alpha \sqsubseteq \overline{\beta}$. *If a maximum/minimum gap constraint has to be satisfied, the "occurrence" relation to hold is* $\alpha \sqsubseteq_c \overline{\beta}$.

**Definition 6. (Sequential pattern mining)** *Given a sequential database and a positive integer* min_sup *(a user-specified threshold), the* sequential mining problem *deals with finding all patterns* $\alpha$ *along with their corresponding supports, such that* $\sigma(\alpha) \geq$ min_sup.

## 2.2 Apriori property and constraints

Also in the FSP problem the Apriori property holds: *all the subsequence of a frequent sequence are frequent.* An FSP constraint C is *anti-monotone* if and only if for any sequence $\beta$ satisfying C, all the subsequences $\alpha$ of $\beta$ satisfy C as well (or, equivalently, if $\alpha$ does not satisfy C, none of the super-sequences $\beta$ of $\alpha$ can satisfy C). Note that the Apriori property is a particular anti-monotone constraint, since it can be restated as *'the constraint on minimum support is anti-monotone'*.

In the problem statement above we have already defined two new constraints besides the minimum

support one: given two *consecutive events* appearing in a sequence, these constraints regards the maximum/minimum valid gap between the *occurrences* of the two events in the various input database sequences.

Consider first the min_gap constraint. Let $\overline{\delta}$ be an input database sequence. If $\beta \sqsubseteq_c \overline{\delta}$, then all its subsequences $\alpha$, $\alpha \preceq \beta$, satisfy $\alpha \sqsubseteq_c \overline{\delta}$. This property holds because $\alpha \preceq \beta$ implies that the gaps between the events of $\alpha$ result "not shorter" than the gaps relative to $\beta$. Hence we can deduce that the min_gap constraint is an anti-monotone constraint.

Conversely, if the max_gap constraint is considered and $\alpha \preceq \beta \sqsubseteq_c \overline{\delta}$, we do not know whether $\alpha \sqsubseteq_c \overline{\delta}$ holds or not. This is because $\alpha \preceq \beta$ implies that the gap between the events of $\alpha$ may be larger than the gaps relative to $\beta$. For example, if $(A{\rightarrow}B{\rightarrow}C) \sqsubseteq_c \overline{\delta}$, the gaps relative to $A{\rightarrow}C$ (i.e. the gaps between the events $A$ and $C$ in $\delta$) are surely larger than the gaps relative to $A{\rightarrow}B$ and $B{\rightarrow}C$. So, if the gap between the events $B$ and $C$ is exactly equal to max_gap, the maximum gap constraint can not be satisfied by $A{\rightarrow}C$, i.e. $A{\rightarrow}C \not\sqsubseteq_c \overline{\delta}$. Hence, we can conclude that, using this definition of sub/super-sequence based on $\preceq$, the max_gap constraint is not anti-monotonic.

## 2.3 Contiguous sequences

We have shown that the property *'$\beta$ satisfies max_gap constraint'* does not propagate to all subsequences $\alpha$ of $\beta$ ($\alpha \preceq \beta$). Nevertheless we can introduce a new definition of subsequence that allows such inference to hold.

**Definition 7. (Contiguous subsequence)** *Given a sequence $\beta = (\beta_1{\rightarrow}...{\rightarrow}\beta_m)$ and a subsequence $\alpha = (\alpha_1{\rightarrow}...{\rightarrow}\alpha_n)$, $\alpha$ is a* contiguous subsequence *of $\beta$, denoted as $\alpha \precsim \beta$, if one of the following holds:*

1. *$\alpha$ is obtained from $\beta$ by dropping an item from either $\beta_1$ or $\beta_m$;*
2. *$\alpha$ is obtained from $\beta$ by dropping an item from $\beta_i$, where $|\beta_i| \geq 2$;*
3. *$\alpha$ is a contiguous subsequence of $\alpha'$, and $\alpha'$ is a contiguous subsequence of $\beta$.*

Note that during the derivation of a contiguous subsequence $\alpha$ from $\beta$, middle events of $\beta$ cannot be removed, so that the gaps between events are preserved. Therefore, if $\overline{\delta}$ is an input database sequence and $\beta \sqsubseteq_c \overline{\delta}$, and $\alpha \precsim \beta$, then $\alpha \sqsubseteq_c \overline{\delta}$ is satisfied in presence of max_gap constraints.

**Lemma 8.** *If we use the concept of* contiguous subsequence *($\precsim$), the maximum gap constraint becomes anti-monotone as well. So, if $\beta$ is a frequent sequential pattern that satisfies the max_gap constraint, then every $\alpha$, $\alpha \precsim \beta$, is frequent and satisfies the same constraint.*

**Definition 9. (Prefix/Suffix subsequence)** *Given a sequence $\alpha = (\alpha_1{\rightarrow}...{\rightarrow}\alpha_n)$ of length $k = |\alpha|$, let $(k-1)$-prefix($\alpha$) ($(k-1)$-suffix($\alpha$)) be the sequence obtained from $\alpha$ by removing the first (last) item of the event $\alpha_1$ ($\alpha_n$). We can say that an item is the first/last one of an event without ambiguity, due to the lexicographic order of items within events.*

*We can now recursively define a generic n-prefix($\alpha$) in terms of the (n+1)-prefix($\alpha$). The n-prefix($\alpha$) is obtained by removing the first (last) item of the first (last) event appearing in the $(n+1)$-prefix($\alpha$). A generic n-suffix($\alpha$) can be defined similarly.*

*It is worth noting that a prefix/suffix of a sequence $\alpha$ is a particular contiguous subsequence of $\alpha$, i.e. n-prefix($\alpha$) $\precsim \alpha$ and n-suffix($\alpha$) $\precsim \alpha$.*

## 2.4 Constraints enforcement

Algorithms solving the FSP problems usually search for $\mathcal{F}_k$ exploiting in some way the knowledge of $\mathcal{F}_{k-1}$. The enforcement of anti-monotone constraints can be pushed deep into the mining algorithm, since patterns not satisfying an anti-monotone constraint C can be discarded immediately, with no alteration to the algorithm completeness (since their super-patterns do not satisfy C too).

More importantly, the anti-monotone constraint C is used during the generation of candidates. Remember that, according to the Apriori definition, a $k$-sequence $\alpha$ can be a "candidate" to include in $\mathcal{F}_k$ only if all of its $(k-1)$-subsequences result to be included in $\mathcal{F}_{k-1}$.

In our case, we will use the $\precsim$ relation to support the notion of subsequence, in order to ensure that *all the contiguous $(k-1)$-subsequences* of $\alpha \in \mathcal{F}_k$ will belong to $\mathcal{F}_{k-1}$. Note that if we used the general notion of subsequence ($\preceq$), the number of the $(k-1)$-subsequences of $\alpha$ should be $k$. Each of them could be obtained by removing a distinct item from one of the events of $\alpha$. Conversely, since we have to use the contiguous subsequence relation ($\precsim$), the number of contiguous $(k-1)$-subsequences of $\alpha$ may be less than $k$: each of them can be obtained by removing a single item only from particular events in $\alpha$, e.g. items belonging to the starting/ending event of $\alpha$.

In practice, each candidate $k$-sequences can simply be generated by combining a single pair of its

contiguous $(k-1)$-subsequences in $\mathcal{F}_{k-1}$. We can now discuss the pattern generation phase of SPADE, which does not use the concept of contiguous subsequences, and thus does not work with max_gap constraints. SPADE generates a candidate $k$-sequence from a pair of frequent $(k-1)$-subsequences that share a common $(k-2)$-prefix. For example, $\alpha = A{\rightarrow}B{\rightarrow}C{\rightarrow}D$ is obtained by combining the two subsequences $A{\rightarrow}B{\rightarrow}C$ and $A{\rightarrow}B{\rightarrow}D$, which share the 2-prefix $A{\rightarrow}B$. Unfortunately, $A{\rightarrow}B{\rightarrow}D$ is not a contiguous subsequence of $\alpha$. This implies that, even if $\alpha$ could be frequent and satisfy a given max_gap constraint, i.e. $\alpha \in \mathcal{F}_4$, its subsequence $A{\rightarrow}B{\rightarrow}D$ could not have been included in $\mathcal{F}_3$ as not satisfying the same max_gap constraint. In other words, SPADE might loose candidates and relative frequent sequences. cSPADE overcomes this limit by exactly using the contiguous subsequence concept. $\alpha = A{\rightarrow}B{\rightarrow}C{\rightarrow}D$ is now obtained from $A{\rightarrow}B{\rightarrow}C$ and $C{\rightarrow}D$, i.e. by combining the $(k-1)$-prefix and the 2-suffix of $\alpha$. It is straightforward to see that both the $(k-1)$-prefix and 2-suffix of $\alpha$ are *contiguous subsequences* of $\alpha$. Unfortunately, the need of using contiguous subsequences to guarantee antimonotonicity under the max_gap constraint partially destroys the prefix-class equivalence self-inclusion of SPADE, which ensures high locality and low memory requirement. While each prefix-class is mined, cSPADE also needs to maintain $\mathcal{F}_2$ in the main memory, since it uses 2-suffixes to extend frequent $(k-1)$-sequences.

The candidate generation method adopted by CCSM was inspired by GSP [9], which is also based on the contiguous subsequence concept. We generate a candidate $k$-sequence $\alpha$ from a pair of frequent $(k-1)$-sequences, that share with $\alpha$ either a $(k-2)$-prefix or a $(k-2)$-suffix. It easy to see that both these frequent $(k-1)$-sequences are contiguous subsequences of $\alpha$.

Note that the number of candidates generated by cSPADE can be larger than the ones generated by CCSM/GSP. We show this with an example. Suppose that $A \rightarrow B \rightarrow C \in \mathcal{F}_3$, and that the only frequent 3-sequence having prefix $B \rightarrow C$ is $B \rightarrow C \rightarrow D$. CCSM directly combine these two 3-sequence to obtain a single potentially frequent 4-sequence $A \rightarrow B \rightarrow C \rightarrow D$. Conversely, cSPADE tries instead to extend $A \rightarrow B \rightarrow C$ with all the 2-sequences in $\mathcal{F}_2$ that start with $C$. In this way, cSPADE might generate a lot of candidate, even if, due to our hypotheses, the only candidate that have chances to be frequent is $A \rightarrow B \rightarrow C \rightarrow D$.

# 3 The CCSM algorithm

Similarly to GSP, CCSM visits level-wise and bottom-up the lattice of the frequent sequential patterns, building at each iteration $\mathcal{F}_k$, the set of all frequent $k$-sequences.

CCSM starts with a count-based phase that mines a horizontal database, and extracts $\mathcal{F}_1$ and $\mathcal{F}_2$. During this phase the database is scanned, and each input sequence is checked against a set of candidate sequences. If the input sequence contains a candidate sequence, the counter associated with the candidate is incremented accordingly. At the end of this count-based phase, the pruned horizontal database is transformed into a vertical one, so that our *intersection-based phase* can start. Thereinafter, when a candidate $k$-sequence is generated from a pair of frequent $(k-1)$-patterns, its support is computed on-the-fly using items idlist intersections. This happens by joining the atomic idlists (stored in the vertical database) that are associated with the frequent items in $\mathcal{F}_1$, as well as several previously computed *intermediate idlists* that are found in a *cache*.

In order to describe how the intersection-based phase works, it is necessary to discuss how candidates are generated, how idlists are represented and joined, and how CCSM idlist cache is organized.

**Candidate generation.** At iteration $k$, we generate the candidate $k$-sequences starting from the frequent $(k-1)$-sequences in $\mathcal{F}_{k-1}$. For each $f \in \mathcal{F}_{k-1}$, we generate candidate $k$-sequences by merging $f$ with every $f' \in \mathcal{F}_{k-1}$ such that $(k-2)$-suffix$(f) = (k-2)$-prefix$(f')$. For example, $f: BD{\rightarrow}B$ is extended with $f': D{\rightarrow}B{\rightarrow}B$ to generate the candidate 4-sequence $BD{\rightarrow}B{\rightarrow}B$. Note that by construction, $f$ and $f'$ are *contiguous subsequences* of the new candidate.

To make more efficient the search in $\mathcal{F}_{k-1}$ for pairs of sequences $f$ and $f'$ that share a common suffix/prefix, we aggregate and link the various *groups* of sequences in $\mathcal{F}_{k-1}$.

Figure 1 illustrates the generation of the candidate 4-sequences starting from $\mathcal{F}_3$. On the left-hand and on the right-hand side of the figure two copies of the 3-sequences in $\mathcal{F}_3$ are shown. These sequences are lexicographically ordered with respect either to their 2-suffixes or 2-prefixes. Moreover, sequences sharing the same suffix/prefix are grouped (this is represented by circling each aggregation/partition with dotted boxes). For example, a partition appearing on the left side is $\{BD \rightarrow B,\ D \rightarrow D \rightarrow B\}$. If two partitions that appear on the opposite sides share a common contiguous 2-subsequence (2-suffix = 2-prefix),

they are also linked together. For instance, two linked partition are $\{BD \rightarrow B,\ D \rightarrow D \rightarrow B\}$ (on the left), and $\{D \rightarrow BD,\ D \rightarrow B \rightarrow B\}$ (on the right). Due to the sharing of suffix/prefix within and between linked partition, we can obviously save memory to represent $\mathcal{F}_3$.

The linked partitions of frequent sequential patterns are the only ones we must combine to generate all the candidates. In the middle of Figure 1 we show the candidates generated for this example. Candidates than do not result frequent are dashed boxed, while the frequent ones are indicated with solid line boxes. Note that, before passing to the next pair, we first generate all the candidates from the current pair of linked partitions. The order in which candidates are generated enhances temporal locality, because the same prefix/suffix is encountered several times in consecutively generated candidates. Our caching system takes advantage of this locality, storing and reusing intermediate idlist joins.

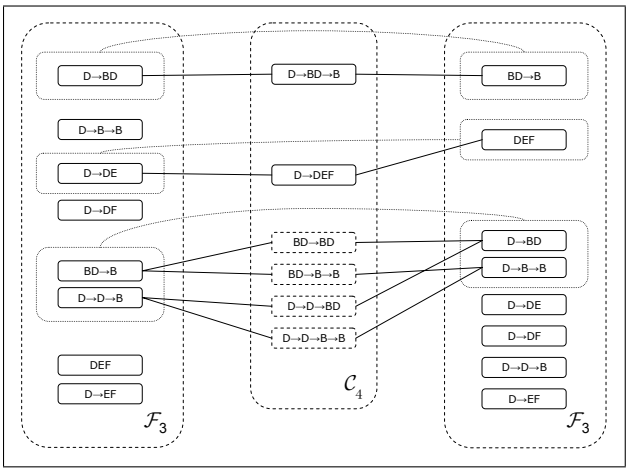


Figure 1: CCSM candidate generation.

**Idlist intersection.** To determine the support of a candidate $k$-sequence $p$, we have first to produce the associated idlist $L(p)$. Its support will correspond to the the number of distinct *sid* values contained in $L(p)$.

To produce $L(p)$, we have to join the idlists associated with two or more subsequences of $p$. If both $L(p'_1)$ and $L(p'_2)$ are available, where $p'_1$ are $p'_2$ are the two contiguous subsequences whose combination produces $p$, $L(p)$ can be generated very efficiently through a 2-way intersection: $L(p) = L(p'_1) \cap L(p'_2)$. Otherwise we have to intersect idlists associated with smaller subsequences of $p$. The limit case is a $k$-way intersection, when we have to intersect atomic idlists associated with single items.

As an example of a $k$-way intersection, consider the candidate 3-sequence $A \rightarrow B \rightarrow C$. Our vertical database stores $L(A)$, $L(B)$ and $L(C)$, which can be joined to produce $L(A \rightarrow B \rightarrow C)$. Each atomic list stores $(sid, eid)$ pairs, i.e. the temporal occurrences $(eid)$ of the associated item within the original input sequences $(sid)$. When $L(A)$, $L(B)$ and $L(C)$ are joined, we search for all occurrences of $A$ followed by an occurrence of $B$, and then, using the intermediate result $L(A \rightarrow B)$, for occurrences of $C$ after $A \rightarrow B$. If a maximum or minimum gap constraint must be satisfied, it is also checked on the associated timestamps $(eid$s$)$.

Note that in this case we have generated the pattern $A \rightarrow B \rightarrow C$ by extending the pattern from left to right. An important question regards what information has to be stored along with the intermediate list $L(A \rightarrow B)$. We can simply show that, if we extend the pattern from left to right, the only information needed for this operation is those related to timestamps associated with the *last* item/event of the sequence. With respect to $L(A \rightarrow B)$, this information consists in the list of $(sid, eid)$ pairs of the $B$ event. Each pair indicates that an occurrence of the specified sequential pattern occurs in the input sequence *sid*, *ending* at time *eid*.

On the other hand, if we generate the sequence by extending it from right to left, the intermediate sequence should be $B \rightarrow C$, but the information to store in $L(B \rightarrow C)$ should be related to the *first* item/event of the sequence $(B)$. In this case, each $(sid, eid)$ pair stored in the idlist should indicate that an occurrence of the specified sequential pattern exists in input sequence *sid*, *starting* at time *eid*.

Consider now that we use a cache to store intermediate sequences and associated idlists. In order to improve cache reuse, we want to exploit cached sequences to extend other sequences from left to right and vice versa. Therefore the lists of pairs $(sid, eid)$ should be replaced with a lists of terns $(sid, first\_eid, last\_eid)$, indicating that an occurrence of the specified sequential pattern occurs in input sequence *sid*, *starting* at time $first\_eid$ and *ending* at time $last\_eid$.

Finally, note that two types of idlist join are possible: equality join (denoted as $\cap_e$) and temporal join (denoted as $\cap_t$). The first is the usual set-intersection, and is used when we search for occurrences of one item appearing simultaneously with the last item of the current sequence: for example, $L(A \rightarrow BC) = L(A \rightarrow B) \cap_e L(C))$. Temporal

join is instead an ordering-aware intersection operation, which may also check whether the minimum and maximum gap constraints are satisfied. Consider the join of the example above, i.e. $L(A{\rightarrow}B{\rightarrow}C) = L(A{\rightarrow}B) \cap_t L(C)$). The result of this join is obtained from $L(C)$ by discarding all its pairs $(sid_2, eid_2)$ with non-matching $sid_1$'s in the first idlist ($L(A{\rightarrow}B{\rightarrow}C)$), or with a matching $sid_1$ that is not associated with any $eid_1$ smaller than $eid_2$.

More formal definitions of the two base cases (lists of pairs) for equality join, and (min gap, max gap) constraint-enforcing temporal join are shown below:

$$L_1 \cap_e L_2 = \{(sid_2, eid_2) \in L_2 | (\exists (sid_1, eid_1) \in L_1) \\ (sid1 = sid2 \wedge eid_1 = eid_2)\}$$

$$L_1 \cap_t L_2 = \{(sid_2, eid_2) \in L_2 | (\exists (sid_1, eid_1) \in L_1) \\ (sid_1 = sid_2 \wedge eid_1 < eid_2 \wedge \\ minGap \leq |eid_2 - eid_1| \leq max\_Gap)\}$$

**Idlist caching.** Our $k$-way intersection method can be improved using a cache of $k$ idlists. Figure 2 shows how our caching strategy works: the table represents the status of the cache after the idlist associated with sequence $A{\rightarrow}A{\rightarrow}BC{\rightarrow}D$ has been computed. Each cache entry is numbered and contains two values: a sequence and its idlist. Each sequence entry $i$ is obtained from entry $(i-1)$ by appending an item. In a similar way, the associated idlist is the result of a join between the previous cached idlist and the idlist associated with the last appended item. When a new sequence is generated, the cache is searched for a common prefix and the associated idlist. If a common prefix is found, CCSM reuses the associated idlist, and rewrite subsequent cache lines. Considering the example of Figure 2, if the candidate $A{\rightarrow}A{\rightarrow}BF$ is then generated, the third cache line corresponding to the common prefix $A{\rightarrow}A{\rightarrow}B$ will be reused. In this way, the support of $A{\rightarrow}A{\rightarrow}BF$ can be computed by performing a single equality join between the idlist in line 3 and $L(F)$. The result of this join is written in line 4 for future reuse.

Since the cache contains all the prefixes of the current sequence along with the associated idlists, reuse is optimal when candidate sequences are generated in lexicographic order. Furthermore, as idlist length (and join cost) decreases as sequence length increases, the joins saved by exploiting the cached idlists are the most expensive ones.

The combined effect of cache use and candidate generation is illustrated in Figure 3. On the left-hand side a fragment of the lists of the linked partitions sharing a common infix is shown. The right-hand side of the Figure illustrates instead how candidates are generated. First we consider the $Partition(FG{\rightarrow}A)$, i.e. the set of sequences sharing the prefix/suffix $FG{\rightarrow}A$. $L(FG{\rightarrow}A)$ is processed first, using the cache as described before. $L(A)$ and $L(B)$ are then joined *left to right* with $L(FG{\rightarrow}A)$ to obtain $L(FG{\rightarrow}A{\rightarrow}A)$ and $L(FG{\rightarrow}A{\rightarrow}B)$. Finally we join *right to left* the lists so obtained with $L(A)$ and $L(C)$ to produce the lists associated with all the possible candidates. When $Partition(FG{\rightarrow}A)$ has been processed, all the intermediate idlists except those stored in cache are discarded, and the next $Partition(FG{\rightarrow}B)$ is processed. The cache currently contains $L(FG{\rightarrow}A)$ and all its intermediate idlists, so that $L(FG)$ can be reused for computing $L(FG{\rightarrow}B)$. Since partitions are ordered with respect to the common infix, similar reuses are very frequent.

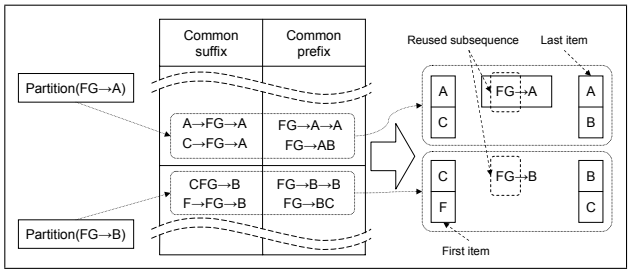


Figure 3: CCSM idlist reuse.

Figure 4 shows the efficacy of CCSM caching strategy. The plots report the actual number of intersection operations performed using 2-ways, pure k-ways and CCSM cached k-ways intersection methods while mining two synthetic datasets. As it can be seen, our small cache is very effective since it allows to save a lot of intersection operations over a pure k-ways method, although memory requirements are significantly lower than those deriving from the adoption of a pure 2-ways intersection method.

# 4 Experimental evaluation

In order to evaluate the performances of the CCSM algorithm, we conducted several tests on a linux box equipped with a 450MHz Pentium II processor, 512MB of RAM and a IDE HD. The datasets used were CS11, and CS21, two synthetic datasets generated with the publicly available IBM quest dataset generator. In particular, the datasets contain $100,000$ customer sequences composed in the average

| | Cached Sequence | Cached Idlist |
|---|---|---|
| 1 | $A$ | $L(A)$ |
| 2 | $A \rightarrow A$ | $L(A) \cap_t L(A)$ |
| 3 | $A \rightarrow A \rightarrow B$ | $[L(A) \cap_t L(A)] \cap_t L(B)$ |
| 4 | $A \rightarrow A \rightarrow BC$ | $[[L(A) \cap_t L(A)] \cap_t L(B)] \cap_e L(C)$ |
| 5 | $A \rightarrow A \rightarrow BC \rightarrow D$ | $[[[L(A) \cap_t L(A)] \cap_t L(B)] \cap_e L(C)] \cap_t L(D)$ |

Figure 2: Example of cache usage.

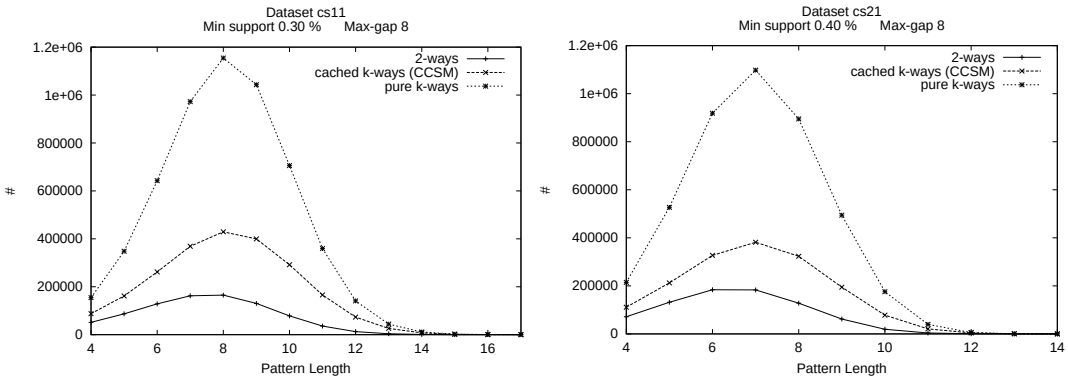


Figure 4: Number of intersection operations actually performed using 2-ways, pure k-ways and cached k-ways intersection methods while mining two synthetic datasets.

of 10 (CS11) and 20 (CS21) transactions of average length 5. The other parameters $N_s$, $N_i$, $N$, $I$ used to generate the maximal sequences of average size $S = 4$ (CS11) and $S = 8$ (CS21), were set to 5000, 25000, 10000 and 2.5, respectively. Note that these values are exactly the same as those used to generate the synthetic datasets in [9, 10, 11]. Figure 5 plots the number of frequent sequences found in datasets CS11 and CS21 as a function of the pattern length for different values of the max_gap constraint. As expected, the number of frequent sequences is maximum when no max_gap constraint is imposed, while it decreases rapidly for decreasing values of the max_gap constraint.

In order to assess the relative performance of our algorithm, we compared its running times with the ones obtained under the same testing conditions by cSPADE (we acknowledge Prof. M.J. Zaki for kindly providing us cSPADE code) [10, 11].

Figure 6 reports the total execution times of CCSM and cSPADE on datasets CS11 and CS21 as a function of the max_gap value. In the tests conducted with cSPADE we tested different configurations of the command line options available to specify the number of partitions into which the dataset has to be split (`-e #`, default no partitioning), and the maximum amount of memory available to the application (`-m #`, default 256MB).

From the plots we can see that while on the CS11 dataset performances of the two algorithms are comparable, on the CS21 dataset CCSM remarkably outperforms cSPADE for large values of max_gap, while cSPADE is faster when max_gaps are small. This holds because for large values of max_gaps, the actual number of frequent sequences is large (see Figure 5), and cSPADE has to perform a lot of intersections between relatively long lists belonging to $\mathcal{F}_2$. CCSM on the other hand, reuses in this case several intersections found in the cache. Since execution times increase rapidly for increasing values of max_gaps, we think that the behavior of CCSM is in general preferable over cSPADE one.

The same considerations can be done looking at the plots reported in Figure 7 that report for a fixed max_gap constraint (max_gap=8), the execution times of CCSM and cSPADE on datasets CS11 and CS21 as a function of the minimum support threshold. The CCSM and cSPADE execution times resulted very similar on the CS11 dataset, while on the CS21 dataset CCSM resulted, for max_gap=8, about twice faster than cSPADE.

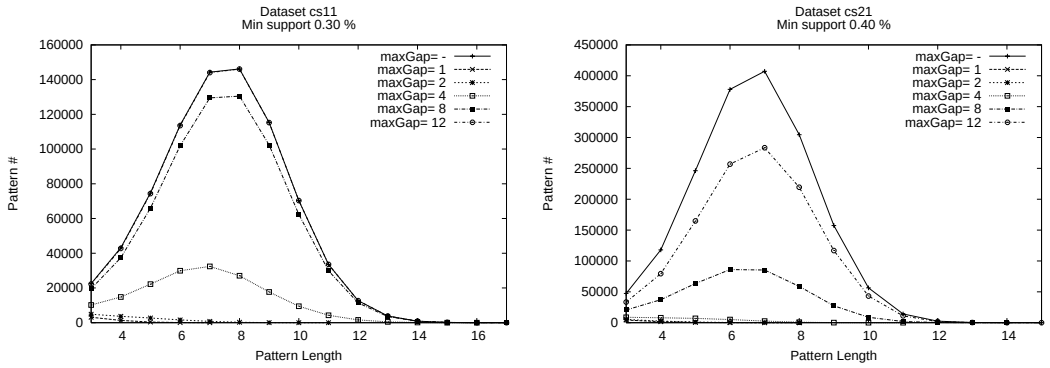


Figure 5: Number of frequent sequences in datasets CS11 (minsup=0.30) and CS21 (minsup=0.40) as a function of the pattern length for different values of the max_gap constraint.

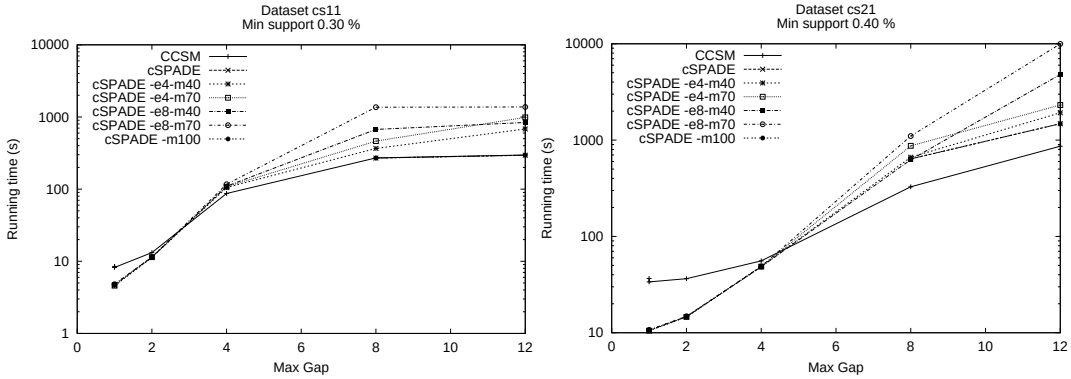


Figure 6: Execution times of CCSM and cSPADE on datasets CS11 (minsup=0.30) and CS21 (minsup=0.40) as a function of the max_gap value.

# 5 Conclusions

We have presented CCSM, a new FSP algorithm that mine temporal databases in the presence of user-defined constraints. CCSM searches for sequential patterns level-wise, and adopts an intersection-based method to determine the support of candidate $k$-sequences. Our $k$-way intersection method follows the promising methodology exploited by the DCI algorithm for FP mining. Each time a candidate $k$-sequence $\alpha$ is generated, its support is determined on the fly by joining the $k$ atomic idlists associated with the frequent items (1-sequences) constituting the candidate. This $k$-way intersection is, however, a limit case of our method. In fact our order of generation of candidate ensures high locality, so that with high probability successively generated candidates share a common subsequence of $\alpha$. A cache is thus used to store the intermediate idlists associated with all the possible prefixes of $\alpha$. When the idlist of another candidate $\beta$ has to be built, we reuse the idlist corresponding to the common subsequence of maximal length. The exploitation of such caching strategy entails a strong reduction in the number of join operations actually performed. Finally, CCSM is able to consider the very challenge max_gap constraint over the sequential pattern extracted. Preliminary experiments conducted on synthetically generated datasets showed that CCSM remarkably outperforms cSPADE when the selectivity of the gap constraint is not high. Since we are conscious that further optimization can be pushed into the code, we consider these results as encouraging. We are now working on these optimizations and on extensions aimed to support other challenging type of constraints. Our final goal is to design an adaptive algorithm that modifies its behavior on the basis of the particular constraints defined by the user.

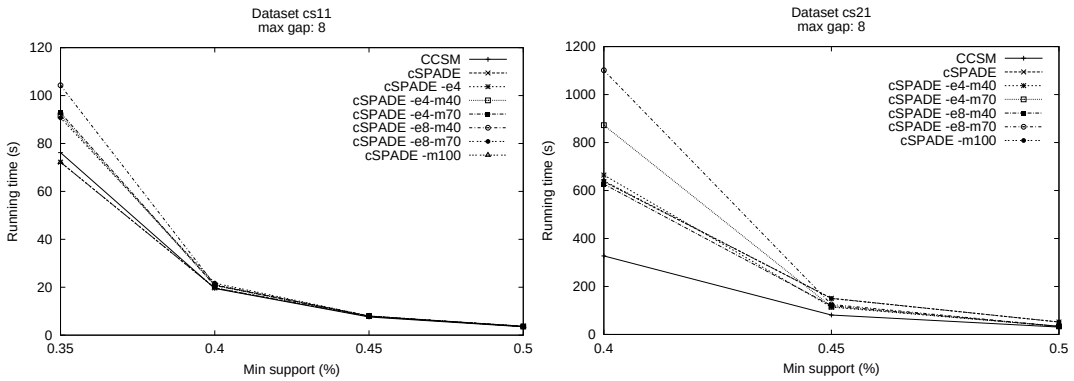


Figure 7: Execution times of CCSM and cSPADE on datasets CS11 and CS21 with a fixed max_gap constraint (max_gap=8) as a function of the minimum support threshold.

# References


[1] R. Agrawal, T. Imielinski, and S. A., *Mining Associations between Sets of Items in Massive Databases*, in Proc. of the ACM-SIGMOD 1993 Int'l Conf. on Management of Data, 1993, pp. 207–216.

[2] R. Agrawal and R. Srikant, *Mining sequential patterns*, in Proc. 11th Int. Conf. Data Engineering, ICDE, P. S. Yu and A. L. P. Chen, eds., IEEE Press, 6–10 1995, pp. 3–14.

[3] J. Ayres, J. Gehrke, T. Yiu, and J. Flannick, *Sequential pattern mining using bitmaps*, in In Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining., 2002.

[4] M. N. Garofalakis, R. Rastogi, and K. Shim, *SPIRIT: Sequential pattern mining with regular expression constraints*, in The VLDB Journal, 1999, pp. 223–234.

[5] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri, *Adaptive and resource-aware mining of frequent sets*, in Proc. of the 2002 IEEE International Conference on Data Mining, ICDM, 2002.

[6] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu, *Prefixspan: Mining sequential patterns by prefix-projected growth*, in ICDE, 2001, pp. 215–224.

[7] J. Pei, J. Han, and W. Wang, *Mining sequential patterns with constraints in large databases*, in Proc. of Proceedings of the 11-th Int. Conf. on Information and Knowledge Management (CIKM 02), 2002, pp. 18–25.

[8] A. Savasere, E. Omiecinski, and S. B. Navathe, *An efficient algorithm for mining association rules in large databases*, in VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland, U. Dayal, P. M. D. Gray, and S. Nishio, eds., Morgan Kaufmann, 1995, pp. 432–444.

[9] R. Srikant and R. Agrawal, *Mining sequential patterns: Generalizations and performance improvements*, in Proc. 5th Int. Conf. Extending Database Technology, EDBT, P. M. G. Apers, M. Bouzeghoub, and G. Gardarin, eds., vol. 1057, Springer-Verlag, 25–29 1996, pp. 3–17.

[10] M. J. Zaki, *Sequence mining in categorical domains: Incorporating constraints*, in CIKM, 2000, pp. 422–429.

[11] ——, *SPADE: An efficient algorithm for mining frequent sequences*, Machine Learning, 42 (2001), pp. 31–60.