Algorithm for sequence mining using gap constraints

Pinkal Shah, Prof. (Dr.) A.K.Dua

[1]*Computer Science, SET Amity University Rajasthan Jaipur, India*
[2]*M.Tech (Computer Science), ASET Amity University Rajasthan Jaipur, India*

Abstract:- The sequence mining problem consists in finding frequent sequential patterns in a database of time-stamped events. Some application domains require limiting the maximum temporal gap between events in the input sequences. However concentration on such constraint is critical for most sequence mining algorithms. In this paper we describe CCSM (Cache-based Con-strained Sequence Miner), a new level-wise algorithm that overcomes the troubles usually related to this kind of con-straints. CCSM adopts an approach based on 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 reuse in future. The reuse of intermediate results involves a surprising reduction in the actual number of join operations performed on idlists. CCSM has been experimentally compared with cSPADE, a state of the art algorithm, on several synthetically generated datasets, achieve better or similar results in most cases.
Categories and Subject Descriptors— Database management: Database Applications - Data Mining.
General Terms— Algorithms, Performance, Experimentation.

Keywords:- Sequential patterns and mining, constraints, intersection, vertical dataset.

I.   INTRODUCTION

The problem of mining frequent sequential patterns was introduced by Agraval and Srikant in [2]. In the following work the same authors discussed the introduction of con-straints on the mined sequences, and proposed GSP [9], a new algorithm dealing with them. In the last years, many new algorithms were given 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 few Frequent Sequential Patterns (FSP) as an abstraction of Frequent Pattern (FP) mining to temporal databases. FP mining aims to find patterns (itemsets) occurring with minimum support within a transactional database (D), whose transactions correlated to collections of items. A pattern is frequent if its support is greater than or equal to a given threshold x%, i.e. if it is set-included in at least x%. Database |D| input transactions, where |D| is the total number of transactions of D. A given database D for the FSP problem is instead composed of a collection of sequences. Each of them correlated to a temporally ordered list of events, where each event is a collection of items (itemsets) occurring concurrently.

The temporal ordering among the events is directed 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 x%, i.e. if it is "contained" in (or it is a subsequence of) at least x%. Database |D| input sequences, where |D| is the number of sequences included in D. To make more natural both problem formulations, we may consider them within the application context of the market basket analysis (MBA). In this context, each transaction (itemsets) occurring in a database D of the FP problem correlates 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, Database D has also to include information about timestamp and customer identity of each basket. The sequences of events included in D correspond to sequences of "baskets" (transactions) purchased by the same customer during individual visits to the market, and the items of a sequential pattern can extent a set of subsequent transactions belonging to the same customer. Thus, while the FP problem is interested in finding intera-transaction patterns, the FSP problem determines inter-transaction sequential patterns. Due to the similarities between the FP and FSP problems, several FP algorithms have

been adapted for mining frequent sequential patterns. Also like FP algorithms, FSP 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-Search) of the search space. At each iteration k, a set of candidate k-sequences of length k is generated, and the dataset, stored in horizontal form, and it gets scanned to count how many times each candidate is contained within each input sequences. Whereas the intersection-based approach, depends on a vertical-layout database, where for each item a appearing in various input sequences we store an *idlist* L(A). The idlist contains information about the identifiers of the input sequences (*sid*) that include A, and the timestamps (*eid*) associated with each occurrence of A. Idlists are thus composed of pairs *(sid, eid)*, and are more complex than the lists of transaction identifiers (*tidlists*) explored by intersection-based FP algorithms. Using an intersection-based method, the support of a candidate is decided by joining lists. In the FP algorithm, tidlists joining is done by means of simple set-intersection. Conversely, idlist joining in FSP intersection-based algorithms explores more complex temporal join operation. SPADE algorithm [11] is the best example 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 searching 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 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 unnatural constraints. Unfortunately, when the constraint is not on the sequence itself but on its occurrences (as in the case 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.

Generally, FSP algorithms that directly deal with user-provided constraints during the mining process are much more efficient, since constraints may involve an effective pruning of candidates, thus resulting in a strong reduction of the computational cost.

Unfortunately, the inclusion of some support-related constraints may require large number of 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 count 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 maintaining 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 blocks. This is accomplished by exploring 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 Zaki's 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 bit more expensive way to generate sequences and join idlists, also maintaining in memory 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 maximum gap constraint. The main addition of CCSM 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 generated idlists.

The *idlist reuse* allowed by cache lead to 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. Here in this 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.

II.   SEQUENTIAL PATTERNS MINING

A.   2.1 Problem Statement

Definition 1. (SEQUENCE OF EVENTS) Let $I = \{i_{1,\ldots,} i_m\}$ be a set of m distinct items. An event (itemset) is a non-empty

subset of I. A sequence is a temporally ordered list of events.

We denote an event as $(j_1, j_{2\ldots} , j_m)$ and a sequence as $(\alpha_1 \rightarrow, . , \rightarrow \alpha_k)$, where each j_i is an item and each α_i is an event ($j_i \in I$ and α_i

$\subseteq I$). 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 ($|x| = \Sigma | \alpha_i |$). 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 un-equivocal the way in which a sequence is written, e.g., we cannot write $QP \rightarrow YXZ$ since the correct way is $PQ \rightarrow XYZ$

This allows us to say, without ambiguity, that the sequence

$P \rightarrow QS$ is a prefix of $P \rightarrow XYZ \rightarrow P$, while $YZ \rightarrow P$ is a suffix. A prefix/suffix of a given sequence is particular subsequences of α (see Def. below).

Definition 2.   (Sequence of Events) A Sequence

$\alpha = (\alpha_1 \rightarrow, \ldots, \rightarrow \alpha_k)$ is contained in a sequence $\beta = (\beta_1 \rightarrow, \ldots, \rightarrow \beta_m)$ (denoted as), if there exist integers $1 \leq i_1 < \ldots <$

$i_k \leq m$ such hat $\alpha_1 \subseteq \beta_{i1,\ldots,} \alpha_k \subseteq \beta_{ik.}$ We also say that is a subsequence of β , and that β is a super-sequence of $\alpha..$

Definition 3. (Sequence Of Events) A temporal database is a collection of input sequences :

$D = \{ \alpha \mid \alpha = (sid, \alpha, eid) \}$,

where sid is a sequence identifier, $\alpha = (\alpha_1 \rightarrow, \ldots, \rightarrow \alpha_k)$ is an event sequence, and eid = $(eid_1, eid_2, \ldots, eid_k)$ is a tuple of unique event identifiers, where each eid_i is the timestamp (occurring time) of event α_i .

DEFINITION 4 . (GAP CONSTRAINED OCCURRENCE OF A SEQUENCE) Let β a given input sequence, whose events $\beta = (\beta_1 \rightarrow, \ldots, \rightarrow \beta_m)$ are time-stamped with $(eid_1, eid_2, \ldots, eid_k)$

. The gap between two consecutive events β_i and β_{i+1} is thus defined as $(eid_{i+1} - eid_i)$.

A sequence $\alpha = (\alpha_1 \rightarrow, \ldots, \rightarrow \alpha_k)$ occurs in β under the minimum gap and maximum gap constraints, denoted as $\alpha \subseteq \beta$, if there

exists integers $1 \leq i_1 \leq \ldots \leq i_k \leq m$ such that $\alpha_1 \subseteq \beta_{i1,\ldots,} \alpha_k \subseteq \beta_{ik.}$ and $\forall_j, 1 < i_1 < \ldots < i_k,$ min_gap $\leq (eid_{ij} - eid_{ij-1}) \leq$ max_gap, where min_gap and max_gap are user specified thresholds.

When no constraints are specified, we denote the occurrence of

α in β as $\alpha \subseteq \beta$. This case is a simpler case of sequence occurrence, since we have that $\alpha \subseteq \beta$ simply if $\alpha \leq \beta$ holds.

Definition 5 . (SUPPORT AND CONSTRAINTS) The support of a sequence pattern α , denoted as $\sigma(\alpha)$, is the number of distinct input sequences β such that $\alpha \subseteq \beta$. If a maximum/minimum gap constraint has to be satisfied, the"\occurrence" relation to hold is $\alpha \subseteq_c \beta$.

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 α along with their corresponding supports, such that $\sigma(\alpha) \geq$ min sup.

B.   *Apriori Property And Constraints*
Also in the FSP problem the Apriori property holds: all the subsequences of a frequent sequence are frequent. An FSP constraint C is anti-monotone if and only if for any sequence β satisfying C, all the subsequences α of β satisfy C as well (or, equivalently, if α does not satisfy C, none of the super-sequences β of α 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 δ be an input database sequence. If $\beta \subseteq_c \delta$, then all its subsequences α, $\alpha \leq \beta$, satisfy $\alpha \subseteq c \delta$. This property holds because $\alpha \leq \beta$ implies that the gaps between the events of α result "not shorter" than the gaps relative to β. Hence we can deduce that the min_gap constraint is an anti-monotone constraint.

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

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

Definition 7. (CONTIGUOUS SUBSEQUENCE) Given a sequence $\beta = (\beta_1 \rightarrow \dots \rightarrow \beta_m)$ and a subsequence $\alpha = (\alpha_1 \rightarrow, \dots, \rightarrow \alpha_n)$, is a contiguous subsequence of β, denoted as $\alpha \leq \beta$, if one of the following holds:

1. α is obtained from β by dropping an item from either β_1 or β_m ;

2. α is obtained from β by dropping an item from β_i , where $| \beta_i | \geq 2$;

3. α is a contiguous subsequence of α', and α' is a contiguous subsequence of β.

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

LEMMA 8. If we use the concept of contiguous subsequence ($\leq$), the maximum gap constraint becomes anti-monotone as well. So, if β is a frequent sequential pattern that satisfies the max gap constraint, then every α, $\alpha \leq \beta$

DEFINITION 9. (PREFIX/SUFFIX SUBSEQUENCE) Given a sequence $\alpha = (\alpha_1 \rightarrow, \ldots, \rightarrow \alpha_n)$ of length $k = |\alpha|$, let (k-1)-prefix(α) ((k-1)-suffix(α)) be the sequence obtained from α by removing the first (last) item of the event α_1 (α_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(α) in terms of the

(n+1)-prefix(α). The n-prefix(α) is obtained by removing the first (last) item of the first (last) event appearing in the $(n + 1)$-prefix(α). A generic n-suffix(α) can be defined similarly. It is worth noting that a prefix/suffix of a sequence α is a Particular contiguous subsequence of α, i.e. n-prefix(α) $\leq \alpha$ and n-suffix(α) $\leq \alpha$.

D. Constraints Enforcement

Algorithms solving the FSP problems usually search for F_k exploiting in some way the knowledge of 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 can be a "Candidate" to include in F_k only if all of its (k-1)-subsequences result to be
Included F_{k-1}.

In our case, we will use the $\leq$ relation to support the notion of subsequence, in order to ensure that all the contiguous (k-1)-subsequences of $\alpha \in F_k$ will belong to F_{k-1}. Note that if we used the general notion of subsequence ($\leq$), the number of the (k-1)-subsequences of α should be k. Each of them could be obtained by removing a distinct item from one of the events of α.

Conversely, since we have to use the contiguous subsequence relation ($\leq$), the number of contiguous (k-1)-subsequences of α may be less than k : each of them can be obtained by removing a single item only from particular events in α, e.g. items belonging to the starting/ending event of α.

In practice, each candidate k-sequence can simply be generated by combining a single pair of its contiguous (k-1)-subsequences in 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, $\alpha = P \rightarrow Q \rightarrow R \rightarrow S$ is obtained by combining the two subsequences $P \rightarrow Q \rightarrow R$ and $P \rightarrow Q \rightarrow S$, which share the 2-preffix $P \rightarrow Q$.

Unfortunately, $P \rightarrow Q \rightarrow S$ is not a contiguous subsequence of α This implies that, even if α could be frequent and satisfy a given max_gap constraint, i.e. $\alpha \in F_4$, its sub-sequence $P \rightarrow Q \rightarrow S$ could not have been included in 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. $\alpha = P \rightarrow Q \rightarrow R \rightarrow S$ is now obtained from $P \rightarrow Q \rightarrow R$ and $R \rightarrow S$, i.e. by combining the (k-1)-prefix and the 2-suffix of α. It is straightforward to see that both the (k-1)-prefix and 2-suffix of α are contiguous subsequences of α.

Unfortunately, the need of using contiguous subsequences to guarantee anti-monotonicity 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 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 α from a pair of frequent (k-1)-sequences, that share with α 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 α.

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 $P \rightarrow Q \rightarrow R \in F_3$, and that the only frequent 3-sequence having prefix $Q \rightarrow R$ is $Q \rightarrow R \rightarrow$

S. CCSM directly combines these two 3-sequences to obtain a single potentially frequent 4-sequence $P \rightarrow Q \rightarrow R \rightarrow S$. Conversely, cSPADE tries instead to ex-tend $A \rightarrow B \rightarrow C$ with all the 2-sequences in F_2 that start with C. In this way, cSPADE might generate a lot of candidates, even if, due to our hypotheses, the only candidate that has chances to be frequent is $P \rightarrow Q \rightarrow R \rightarrow S$.

III.   THE CCSM ALGORITHM

Similarly to GSP, CCSM visits level-wise and bottom-up the lattice of the frequent sequential patterns, building at each iteration F_k, the set of all frequent k-sequences. CCSM starts with a count-based phase that mines a horizontal database, and extracts F_1 and 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 trans-formed 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 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 F_{k-1} . For each $f \in F_{k-1}$, we generate candidate k-sequences by merging f with every f' $\in F_{k-1}$ such that (k-2)-suffix(f) = (k-2)-prefix(f). For example, f : QS$\rightarrow$Q is extended with f ': S$\rightarrow$Q$\rightarrow$Q to generate the candidate 4-sequence QS$\rightarrow$Q$\rightarrow$Q. Note that by construction, f and f ' are *contiguous subsequences* of the new candidate. To make more efficient the search in 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 F_{k-1}.

Figure 1 illustrates the generation of the candidate 4-sequences starting from F_3. On the left-hand and on the right-hand side of the figure two copies of the 3-sequences in F_3 are shown. These sequences are lexicographically ordered with respect either to their 2-suffixesor 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 {QS $\rightarrow$ Q; S $\rightarrow$ S $\rightarrow$ Q}.

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

{QS $\rightarrow$ Q; S $\rightarrow$ S $\rightarrow$ Q} (on the left), and {S $\rightarrow$ QS; S $\rightarrow$ Q $\rightarrow$

Q} (on the right). Due to the sharing of suffix/prefix within and between linked partitions, we can obviously save memory to represent 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.
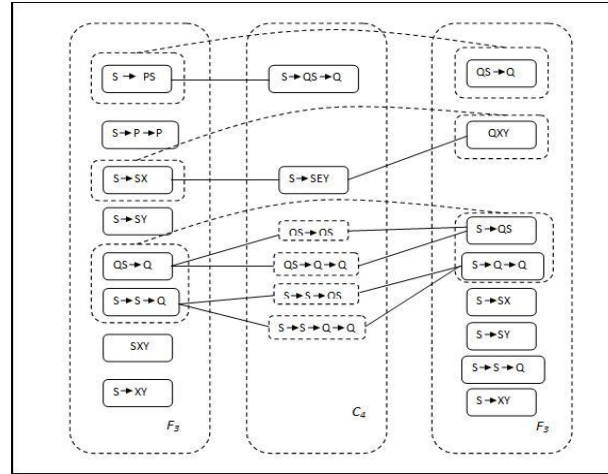
Figure 1: CCSM candidate generation.

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 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 P→Q→R. Our vertical database stores L(P), L(Q) and L(R), which can be joined to produce L(P→Q→R). 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(P), L(Q) and L(R) are joined, we search for all occurrences of A followed by an occurrence of B, and then, using the intermediate result L(P→Q), for occurrences of C after P→Q. If a maximum or minimum gap constraint must be satisfied, it is also checked on the associated timestamps (eids).

Note that in this case we have generated the pattern

P→Q→R by extending the pattern from left to right. An important question regards what information has to be stored along with the intermediate list L(P→Q). 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(P→Q), this information consists in the list of (sid; eid) pairs of the Q 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→C, but the information to store in L(B→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 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 $\cap_t$

Idlist caching. Our k-way intersection method can be for equality join, and (min_gap, max_gap) constraint-

enforcing temporal join are shown below:

$L_{1 \cap_e} L_2 = \{(sid_2, eid_2) \in L_2 \mid (\exists(sid_1, eid_1) \in L_1)$
$(sid1 = sid2 \wedge eid_1 = eid_2)\}$

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 Considering the example of Figure 2, if the candidate P→P→QF is then generated, the third cache line corresponding to the common prefix P→P→Q will be reused. In this way, the support of P→P→QF can be computed by performing a single equality join between the idlist in line 3 and L(F).

when candidate sequences are generated in lexicographic or-der. 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.

s.no	Cached Sequence	Cached Id list
1	P	L(P)
2	P→P	L(P) ∩t L(P)
3	P→P→Q	[L(P) ∩t L(P) ∩t L(Q)
4	P→P→QR	[[L(P) ∩t L(P) ∩t L(Q) ∩e L(R)
5	P→P→QR→S	[[[L(P) ∩t L(P) ∩t L(Q) ∩e L(R) ∩t L(S)

Figure 2: Example of cache usage.

Figure 3: CCSM idlist reuse.

$L_1 \cap_t L_2 = \{(sid_2, eid_2) \in L_2 \; j(\exists(sid_1, eid_1) \in L_1)(sid_1 = sid_2 \wedge eid_1 < eid_2 \wedge$
$min\_gap \le |eid_2 - eid_1| \le max\_gap)\}$

Improved using a cache of k idlists. Figure 2 shows how our 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 rewrites sub). 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(P→QC) = L(P→Q) ∩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(P→Q→R) = L(P→Q) ∩t L(C)). The result of this join is obtained from L(C) by discarding all its pairs (sid₂, eid₂) with non-matching sid₁'s in the first idlist (L(P→Q)), or with a matching sid₁ that is not associated with any eid₁ smaller than eid₂ .

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→A), i.e. the set of sequences sharing the prefix/suffix F G→A. L(FG→A) is processed first, using the cache as described before. L(A) and L(B) are then joined *left to right* with L(FG → A) to obtain L(FG→A→A) and L(FG→P→Q). 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→A) has been processed, all the intermediate idlists except those stored in cache are discarded, and the next Partition

(FG→B) is processed. The cache currently contains L(FG→A) and all its intermediate idlists, so that L(FG) can be reused for computing L(FG→B). Since partitions are ordered with respect to the common infix, similar reuses are very frequent.

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 saving 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.

IV.   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 an 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 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_gap is small. This holds because for large values of max_gap, 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 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_gap, we think that the behavior of CCSM is in general preferable over cSPADE one.
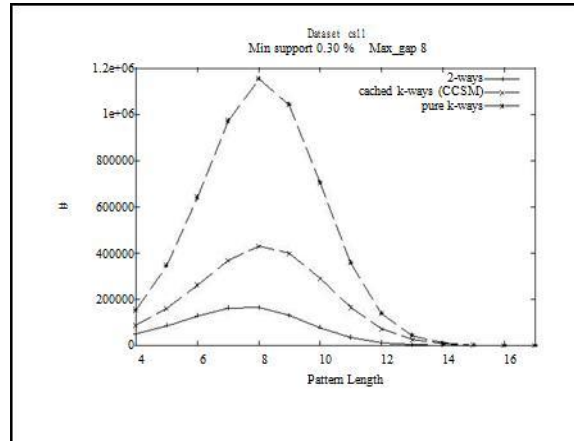
The same considerations can be done looking at the plots reported in Figure 7 that report for a fixed max_gap constraint (max_gap=8), the execution times of CCSM and cSPADE on datasets CS11 and CS21 as a function of the minimum support threshold. The CCSM and cSPADE execution times resulted very similar on the CS11 dataset, while on the CS21 dataset CCSM resulted, for max_gap=8, about twice faster than cSPADE.
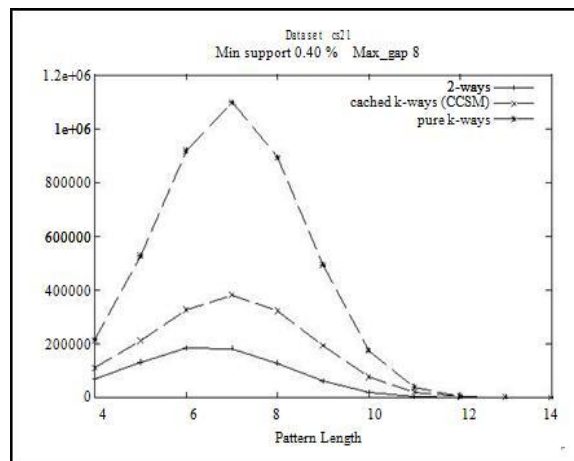
V.   CONCLUSIONS

We have presented CCSM, a new FSP algorithm that mine temporal databases in the presence of user-defined con-straints. CCSM searches for sequential patterns level-wise, and adopts an intersection-based method to determine the support of candidate k-sequences. Each time a candidate k-sequence α is generated, its support is determined on the y 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 α . A cache is thus used to store the intermediate idlists associated with all the possible prefixes of

α. When the idlist of another candidate β 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 challenging 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.
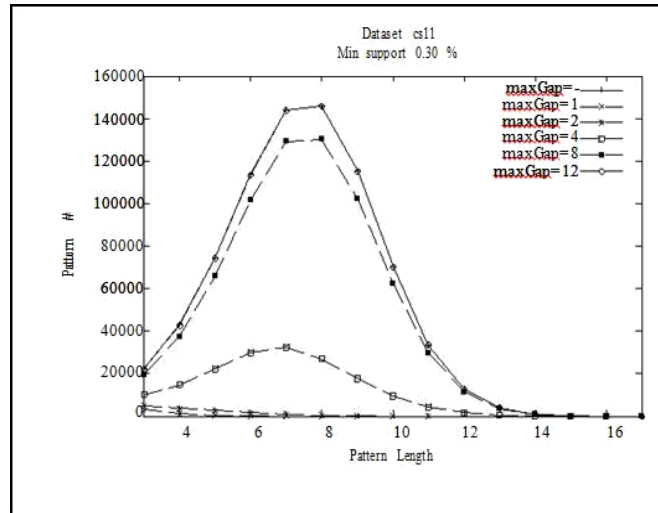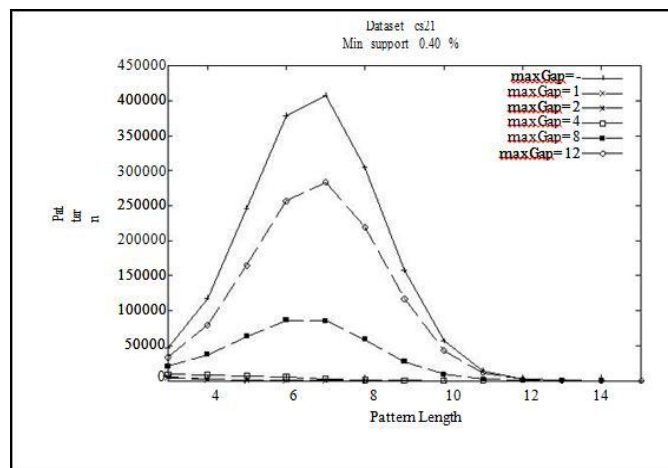
4(a)

4(b)

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.
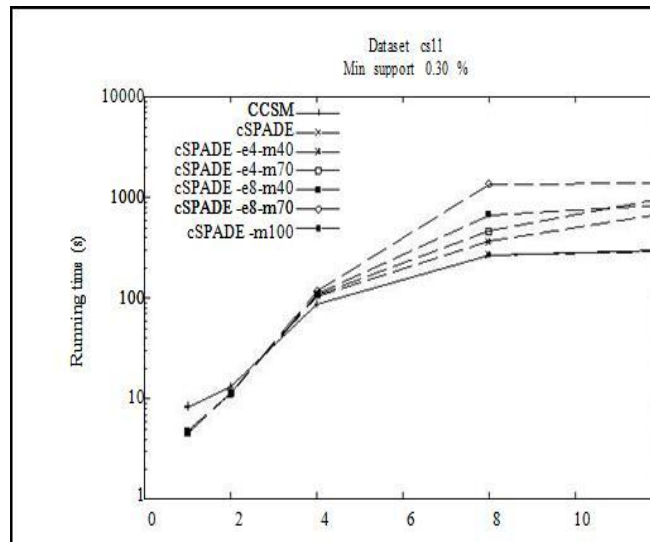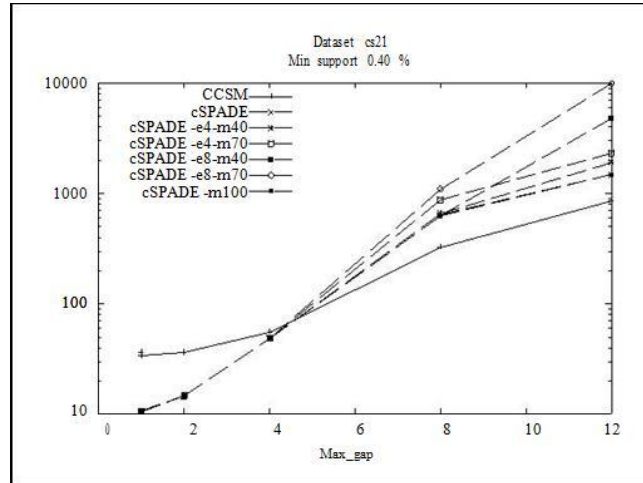
5(a)

5(b)

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.
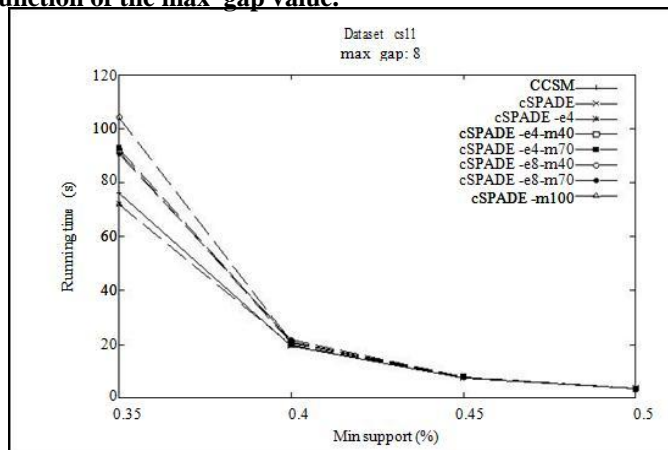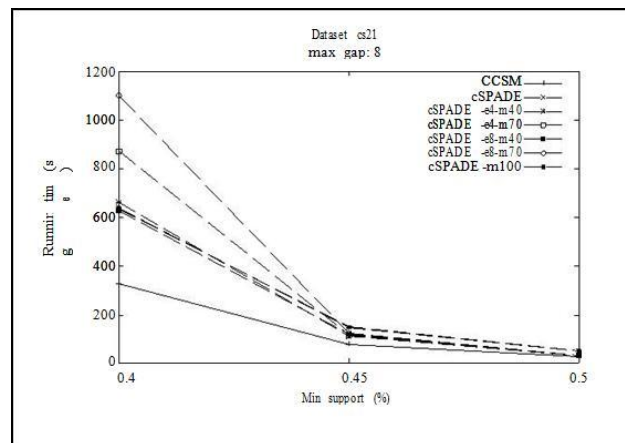

6(a)

6(b)

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.

7(a)

7(b)

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, pages 207{216, 1993.

[2] R. Agrawal and R. Srikant. Mining sequential patterns. In P. S. Yu and A. L. P. Chen, editors, Proc. 11th Int. Conf. Data Engineering, ICDE, pages 3{14. IEEE Press, 6{10 1995.

[3] J. Ayres, J. Gehrke, T. Yiu, and J. Flannick. Sequential pattern mining using bitmaps. 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, pages 223{234, 1999.

[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, pages 215{224, 2001.

[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), pages 18{25, 2002.

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

[9] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In P. M. G. Apers, M. Bouzeghoub, and G. Gardarin, editors, Proc. 5th Int. Conf. Extending Database Technology, EDBT, volume 1057, pages 3{17. Springer-Verlag, 25{29 1996.

[10] M. J. Zaki. Sequence mining in categorical domains: Incorporating constraints. In CIKM, pages 422{429, 2000.

[11] M. J. Zaki. SPADE: An e cient algorithm for mining frequent sequences. Machine Learning, 42(1/2):31{60, 2001.