CAMLS: A Constraint-Based Apriori Algorithm for Mining Long Sequences*

Yaron Gonen, Nurit Gal-Oz, Ran Yahalom, and Ehud Gudes

Department of Computer Science, Ben Gurion University of the Negev, Israel
{yarongon,galoz,yahalomr,ehud}@cs.bgu.ac.il

Abstract. Mining sequential patterns is a key objective in the field of data mining due to its wide range of applications. Given a database of sequences, the challenge is to identify patterns which appear frequently in different sequences. Well known algorithms have proved to be efficient, however these algorithms do not perform well when mining databases that have long frequent sequences. We present CAMLS, Constraint-based Apriori Mining of Long Sequences, an efficient algorithm for mining long sequential patterns under constraints. CAMLS is based on the apriori property and consists of two phases, event-wise and sequence-wise, which employ an iterative process of candidate-generation followed by frequency-testing. The separation into these two phases allows us to: (i) introduce a novel candidate pruning strategy that increases the efficiency of the mining process and (ii) easily incorporate considerations of intra-event and inter-event constraints. Experiments on both synthetic and real datasets show that CAMLS outperforms previous algorithms when mining long sequences.

Keywords: data mining, sequential patterns, frequent sequences.

1 Introduction

The sequential pattern mining task has received much attention in the data mining field due to its broad spectrum of applications. Examples of such applications include analysis of web access, customers shopping patterns, stock markets trends, DNA chains and so on. This task was first introduced by Agrawal and Srikant in [4]: *Given a set of sequences, where each sequence consists of a list of elements and each element consists of a set of items, and given a user-specified min_support threshold, sequential pattern mining is to find all of the frequent subsequences, i.e. the subsequences whose occurrence frequency in the set of sequences is no less than min_support.* In recent years, many studies have contributed to the efficiency of sequential mining algorithms [2,14,4,8,9]. The two major approaches for sequence mining arising from these studies are: **apriori** and **sequence growth**.

* Supported by the IMG4 consortium under the MAGNET program of the Israel ministry of trade and industry; and the Lynn and William Frankel center for computer science.

H. Kitagawa et al. (Eds.): DASFAA 2010, Part I, LNCS 5981, pp. 63–77, 2010.

The **apriori** approach is based on the apriori property, as introduced in the context of association rules mining in [1]. This property states that if a pattern α is not frequent then any pattern β that contains α cannot be frequent. Two of the most successful algorithms that take this approach are GSP [14] and SPADE [2]. The major difference between the two is that GSP uses a horizontal data format while SPADE uses a vertical one.

The **sequence growth** approach does not require candidate generation, it gradually grows the frequent sequences. PrefixSpan [8], which originated from FP-growth [10], uses this approach as follows: first it finds the frequent single items, then it generates a set of projected databases, one database for each frequent item. Each of these databases is then mined recursively while concatenating the frequent single items into a frequent sequence. These algorithms perform well in databases consisting of short frequent sequences. However, when mining databases consisting of long frequent sequences, e.g. stocks values, DNA chains or machine monitoring data, their overall performance exacerbates by an order of magnitude.

Incorporating constraints in the process of mining sequential patterns, is a means to increase the efficiency of this process and to obviate ineffective and surplus output. cSPADE [3] is an extension of SPADE which efficiently considers a versatile set of syntactic constraints. These constraints are fully integrated inside the mining process, with no post-processing step. Pei et al. [7] also discuss the problem of pushing various constraints deep into sequential pattern mining. They identify the prefix-monotone property as the common property of constraints for sequential pattern mining and present a framework (Prefix-growth) that incorporates these constraints into the mining process. Prefix-growth leans on the sequence growth approach [8].

In this paper we introduce CAMLS, a constraint-based algorithm for mining long sequences, that adopts the apriori approach. The motivation for CAMLS emerged from the problem of aging equipment in the semiconductor industry. Statistics show that most semiconductor equipment suffer from significant unscheduled downtime in addition to downtime due to scheduled maintenance. This downtime amounts to a major loss of revenue. A key objective in this context is to extract patterns from monitored equipment data in order to predict its failure and reduce unnecessary downtime. Specifically, we investigated lamp behavior in terms of illumination intensity that was frequently sampled over a long period of time. This data yield a limited amount of possible items and potentially long sequences. Consequently, attempts to apply traditional algorithms, resulted in inadequate execution time.

CAMLS is designed for high performance on a class of domains characterized by long sequences in which each event is composed of a potentially large number of items, but the total number of frequent events is relatively small. CAMLS consists of two phases, event-wise and sequence-wise, which employ an iterative process of candidate-generation followed by frequency-testing. The event-wise phase discovers frequent events satisfying constraints within an event (e.g. two items that cannot reside within the same event). The sequence-wise phase constructs

the frequent sequences and enforces constraints between events within these sequences (e.g. two events must occur one after another within a specified time interval). This separation allows the introduction of a novel pruning strategy which reduces the size of the search space considerably. We aim to utilize specific constraints that are relevant to the class of domains for which CAMLS is designed. Experimental results compare CAMLS to known algorithms and show that its advantage increases as the mined sequences get longer and the number of frequent patterns in them rises.

The major contributions of our algorithm are its novel pruning strategy and straightforward incorporation of constraints. This is what essentially gives CAMLS its high performance, despite of the large amount of frequent patterns that are very common in the domains for which it was designed.

The rest of the paper is organized as follows. Section 2 describes the class of domains for which CAMLS is designed and section 3 provides some necessary definitions. In section 4 we characterize the types of constraints handled by CAMLS and in section 5 we formally present CAMLS. Experimental results are presented in section 6. We conclude by discussing future research directions.

2 Characterization of Domain Class

The classic domain used to demonstrate sequential pattern mining, e.g. [4], is of a retail organization having a large database of customer transactions, where each transaction consists of customer-id, transaction time and the items bought in the transaction. In domains of this class there is no limitation on the total number of items, or the number of items in each transaction.

Consider a different domain such as the stock values domain, where every record consists of a stock id, a date and the value of the stock on closing the business that day. We know that a stock can have only a single value at the end of each day. In addition, since a stock value is numeric and needs to be discretized, the number of different values of a stock is limited by the number of the discretization bins. We also know that a sequence of stock values can have thousands of records, spreading over several years. We classify domains by this sort of properties. We may take advantage of prior knowledge we have on a class of domains, to make our algorithm more efficient. CAMLS aims at the class of domains characterized as follows:

- Large amount of frequent patterns.
- There is a relatively small number of frequent events.

Table 1 shows an example sequence database that will accompany us throughout this paper. It has three sequences. The first contains three events: $(acd), (bcd)$ and b in times 0, 5 and 10 respectively. The second contains three events: a, c and (db) in times 0, 4 and 8 respectively, and the third contains three events: $(de), e$ and (acd) in times 0, 7 and 11 respectively.

In section 6 we present another example concerning the behavior of a Quartz-Tungsten-Halogen lamp, which has similar characteristics and is used for experimental evaluation. Such lamps are used in the semiconductors industry for finding defects in a chip manufacturing process.

Table 1. Example sequence database. Every entry in the table is an event. The first column is the identifier of the sequence. The second column is the time difference from the beginning of the sequence to the occurrence of the event. The third column shows all of the items that constitute the event.

Sequence id (sid)	Event id (eid)	items
1	0	(acd)
1	5	(bcd)
1	10	b
2	0	a
2	4	c
2	8	(bd)
3	0	(cde)
3	7	e
3	11	(acd)

3 Definitions

An *item* is a value assigned to an attribute in our domain. We denote an item as a letter of the alphabet: $a, b, ..., z$. Let $I = \{i_1, i_2, ..., i_m\}$ be the set of all possible items in our domain. An *event* is a nonempty set of items that occurred at the same time. We denote an event as $(i_1, i_2, ..., i_n)$, where $i_j \in I, 1 \leq j \leq n$. An event containing l items is called an l-event. For example, (bcd) is a 3-event. If every item of event e_1 is also an item of event e_2 then e_1 is said to be a *subevent* of e_2, denoted $e_1 \subseteq e_2$. Equivalently, we can say that e_2 is a *superevent* of e_1 or e_2 *contains* e_1. For simplicity, we denote 1-events without the parentheses. Without the loss of generality we assume that items in an event are ordered lexicographically, and that there is a radix order between different events. Notice that if an event e_1 is a proper superset of event e_2 then e_2 is radix ordered before e_1. For example, the event (bc) is radix ordered before the event (abc), and $(bc) \subseteq (abc)$.

A *sequence* is an ordered list of events, where the order of the events in the sequence is the order of their occurrences. We denote a sequence s as $\langle e_1, e_2, ..., e_k \rangle$ where e_j is an event, and e_{j-1} happened before e_j. Notice that an item can occur only once in an event but can occur multiple times in different events in the same sequence. A sequence containing k events is called a k-sequence, in contrast to the classic definition of k-sequence that refers to any sequence containing k items [14]. For example, $\langle (de)e(acd) \rangle$ is a 3-sequence. A sequence $s_1 = \langle e_1^1, e_2^1, ..., e_n^1 \rangle$ is a *subsequence* of sequence $s_2 = \langle e_1^2, e_2^2, ..., e_m^2 \rangle$, denoted $s_1 \subseteq s_2$, if there exists a series of integers $1 \leq j_1 < j_2 < ... < j_n \leq m$ such that $e_1^1 \subseteq e_{j_1}^2 \wedge e_2^1 \subseteq e_{j_2}^2 \wedge ... \wedge e_n^1 \subseteq e_{j_n}^2$. Equivalently we say that s_2 is a *super-sequence* of s_1 or s_2 *contains* s_1. For example $\langle ab \rangle$ and $\langle (bc)f \rangle$ are subsequence of $\langle a(bc)f \rangle$, however $\langle (ab)f \rangle$ is not. Notice that the subsequence relation is transitive, meaning that if $s_1 \subseteq s_2$ and $s_2 \subseteq s_3$ then $s_1 \subseteq s_3$. An *m-prefix* of an n-sequence s is any subsequence of s that contains the first m events of s where $m \leq n$. For example, $\langle a(bc) \rangle$ is a 2-prefix of $\langle a(bc)a(cf) \rangle$. An *m-suffix* of an n-sequence s is any subsequence of s that contains the last m events of s where $m \leq n$. For example, $\langle (cf) \rangle$ is a 1-suffix of $\langle a(bc)a(cf) \rangle$.

A *database of sequences* is an unordered set of sequences, where every sequence in the database has a unique identifier called *sid*. Each event in every sequence is associated with a timestamp which states the time duration that passed from the beginning of the sequence. This means that the first event in every sequence has a timestamp of 0. Since a timestamp is unique for each event in a sequence it is used as an event identifier and called *eid*. The *support* or *frequency* of a sequence s, denoted as $sup(s)$, in a sequence database D is the number of sequences in D that contain s. Given an input integer threshold, called *minimum support*, or *minSup*, we say that s is *frequent* if $sup(s) \geq minSup$. The *frequent patterns mining* task is to find all frequent subsequences in a sequence database for a given minimum support.

4 Constraints

A frequent patterns search may result in a huge number of patterns, most of which are of little interest or useless. Understanding the extent to which a pattern is considered interesting may help in both discarding "bad" patterns and reducing the search space which means faster execution time. Constraints are a means of defining the type of sequences one is looking for.

In their classical definition [4], frequent sequences are defined only by the number of times they appear in the dataset (i.e. frequency). When incorporating constraints, a sequence must also satisfy the constraints for it to be deemed frequent. As suggested in [7], this contributes to the sequence mining process in several aspects. We focus on the following two: **(i) Performance**. Frequent patterns search is a hard task, mainly due to the fact that the search space is extremely large. For example, with d items there are $O(2^{d^k})$ potentially frequent sequences of length k. Limiting the searched sequences via constraints may dramatically reduce the search space and therefore improve the performance. **(ii) Non-contributing patterns**. Usually, when one wishes to mine a sequence database, she is not interested in all frequent patterns, but only in those meeting certain criteria. For example, in a database that contains consecutive values of stocks, one might be interested only in patterns of very profitable stocks. In this case, patterns of unprofitable stocks are considered non-contributing patterns even if they are frequent. By applying constraints, we can disregard non-contributing patterns. We define two types of constraints: **intra-event constraints**, which refer to constraints that are not time related (such as values of attributes) and **inter-events constraints**, which relate to the temporal aspect of the data, i.e. values that can or cannot appear one after the other sequentially. For the purpose of the experiment conducted in this study and in accordance with our domain, we choose to incorporate two inter-event and two intra-events constraints. A formal definition follows.

Intra-event Constraints

- *Singletons* Let $A = \{A_1, A_2, ...A_n\}$ s.t. $A_i \subseteq I$, be the set of sets of items that cannot reside in the same event. Each A_i is called a singleton. For example,

the value of a stock is an item, however a stock cannot have more than one value at the same time, therefore, the set of all possible values for that stock is a singleton.
- *MaxEventLength* The maximum number of items in a single event.

Inter-events Constraints

- *MaxGap* The maximum amount of time allowed between consecutive events. A sequence containing two events that have a time gap which is grater than *MaxGap*, is considered uninteresting.
- *MaxSequenceLength* The maximum number of events in a sequence.

5 The Algorithm

We now present *CAMLS*, a Constraint-based Apriori algorithm for Mining Long Sequences, which is a combination of modified versions of the well known Apriori [4] and SPADE [2] algorithms. The algorithm has two phases, event-wise and sequences-wise, which are detailed in subsections 5.1 and 5.2, respectively. The distinction between the two phases corresponds to the difference between the two types of constraints. As explained below, this design enhances the efficiency of the algorithm and makes it readily extensible for accommodating different constraints. Pseudo code for CAMLS is presented in Algorithm 1.

Algorithm 1. CAMLS
Input $minSup$: minimum support for a frequent pattern.
$maxGap$: maximum time gap between consecutive events.
$maxEventLength$: maximum number of items in every event.
$maxSeqLength$: maximum number of events in a sequence.
D: data set. A: set of singletons.
Output F: the set of all frequent sequences.

procedure CAMLS($minSup, maxGap, maxEventLength$, $maxSeqLength, D, A$)
1: {Event-wise phase}
2: $F_1 \leftarrow$ allFrequentEvents($minSup, maxEventLength, A, D$)
3: $F \leftarrow F_1$
4: {Sequence-wise phase}
5: **for all** k such that $2 \leq k \leq maxSeqLength$ and $F_{k-1} \neq \phi$ **do**
6: $C_k \leftarrow$candidateGen($F_{k-1}, maxGap$)
7: $F_k \leftarrow$prune($F_{k-1}, C_k, minSup, maxGap$)
8: $F \leftarrow F \cup F_k$
9: **end for**
10: **return** F

5.1 Event-Wise Phase

The input to this phase is the database itself and all of the intra-event constraints. During this phase, the algorithm partially disregards the sequential nature of the data, and treats the data as a set of events rather than a set of sequences. Since the data is not sequential, applying the intra-event constraints at this phase is very straightforward. The algorithm is similar to the Apriori algorithm for discovering frequent itemsets as presented in [4].

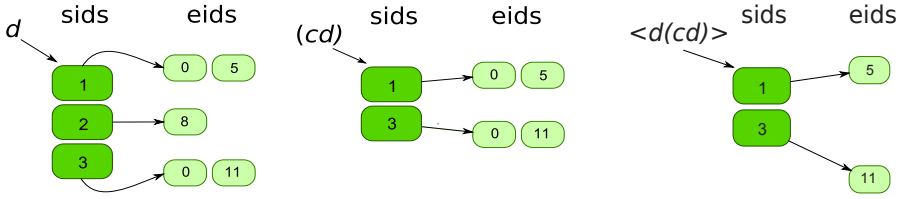
Similarly to Apriori, our algorithm utilizes an iterative approach where frequent $(k-1)$-events are used to discover frequent k-events. However, unlike Apriori, we calculate the support of an event by counting the number of sequences that it appears in rather than counting the number of events that contain it. This means that the appearance of an event in a sequence increases its support by one, regardless of the number of times that it appears in that sequence.

Denoting the set of frequent k-events by L_k (referred to as frequent k-itemsets in [4]), we begin this phase with a complete database scan in order to find L_1. Next, L_1 is used to find L_2, L_2 is used to find L_3 and so on, until the resulting set is empty, or we have reached the maximum number of items in an event as defined by *MaxEventLength*. Another difference between Apriori and this phase lies in the generation process of L_k from L_{k-1}. When we join two $(k-1)$-events to generate a k-event candidate we need to check whether the added item satisfies the rest of the intra-event constraints such as Singletons.

The output of this phase is a radix ordered set of all frequent events satisfying the intra-event constraints, where every event e_i is associated with an *occurrence index*. The occurrence index of the frequent event e_i is a compact representation of all occurrences of e_i in the database and is structured as follows: a list l_i of sids of all sequences in the dataset that contain e_i, where every sid is associated with the list of eids of events in this sequence that contain e_i. For example, Figure 1a shows the indices of events d and (cd) based on the example database of Table 1. We are able to keep this output in main memory, even for long sequences, due to the nature of our domain in which the number of frequent events is relatively small.

Since the support of e_i equals the number of elements in l_i, it is actually stored in e_i's occurrence index. Thus, support is obtained by querying the index instead of scanning the database. In fact, the database scan required to find L_1 is actually the only single scan of the database. Throughout the event-wise phase, additional scans are avoided by keeping the occurrence indices for each event. The *allFrequentEvents* procedure in line 2 of Algorithm 1 is responsible for executing the event-wise phase as described above.

Notice that for mining frequent events there are more efficient algorithms than Apriori, for example FP-growth [10], however, our tests show that the event-wise phase is negligible compared to the sequence-wise phase, so it has no real impact on the running time of the whole algorithm.



(a) The occurrence indices of events d and (cd) from the example database of Table 1. Event d occurs in sequence 1 at timestamps 0 and 5, in sequence 2 at timestamp 8 and in sequence 3 at timestamps 0 and 11. Event (cd) occurs in sequence 1 at timestamps 0 and 5 and in sequence 3 at timestamps 0 and 11.

(b) Example of the occurrence index for sequence $< d(cd) >$ generated from the intersection of the indices of sequences $< d >$ and $< (cd) >$.

Fig. 1. Example of occurrence index

5.2 Sequence-Wise Phase

The input to this phase is the output of the previous one. It is entirely temporal-based, therefore applying the inter-events constraints is straightforward. The output of this phase is a list of frequent sequences satisfying the inter-events constraints. Since it builds on frequent events that satisfied the intra-event constraints, this output amounts to the final list of sequences that satisfy the complete set of constraints.

Similarly to SPADE [2], this phase of the algorithm finds all frequent sequences by employing an iterative candidate generation method based on the apriori property. For the k^{th} iteration, the algorithm outputs the set of frequent k-sequences, denoted as F_k, as follows: it starts by generating the set of all k-sequence candidates, denoted as C_k from F_{k-1} (found in the previous iteration). Then it prunes candidates that do not require support calculation in order to determine that they are non-frequent. Finally, it calculates the remaining candidates' support and removes the non-frequent ones.

The following subsections elaborate on each of the above steps. This phase of the algorithm is a modification of SPADE in which the candidate generation process is accelerated. Unlike SPADE, that does not have an event-wise phase and needs to generate the candidates at an item level (one item at a time), our algorithm generates the candidates at an event level, (one event at a time). This approach can be significantly faster because it allows us to use an efficient pruning method(section 5.2) that would otherwise not be possible.

Candidate Generation: We now describe the generation of C_k from F_{k-1}. For each pair of sequences $s_1, s_2 \in F_{k-1}$ that have a common $(k-2)$-prefix, we conditionally add two new k-sequence candidates to C_k as follows: (i) if s_1 is a generator (see section 5.2), we generate a new k-sequence candidate by concatenating the 1-suffix of s_2 to s_1; (ii) if s_2 is a generator, we generate a new k-sequence candidate by concatenating the 1-suffix of s_1 to s_2. It can be easily proven that if F_{k-1} is radix ordered then the generated C_k is also

radix ordered. For example, consider the following two 3-sequences: $\langle(ab)c(bd)\rangle$ and $\langle(ab)cf\rangle$. Assume that these are frequent generator sequences from which we want to generate 4-sequence candidates. The common 2-prefix of both 3-sequences is $\langle(ab)c\rangle$ and their 1-suffixes are $\langle(bd)\rangle$ and $\langle f\rangle$. The two 2-sequences we get from the concatenation of the 1-suffixes are $\langle(bd)f\rangle$ and $\langle f(bd)\rangle$. Therefore, the resulting 4-sequence candidates are $\langle(ab)cf(bd)\rangle$ and $\langle(ab)c(bd)f\rangle$. Pseudo code for the candidate generation step is presented in Algorithm 2.

Algorithm 2. Candidate Generation
Input F_{k-1}: the set of frequent $(k-1)$-sequences.
Output C_k: the set of k-sequence candidates.

procedure candidateGen(F_{k-1})
1: **for all** sequence $s_1 \in F_{k-1}$ **do**
2: **for all** sequence $s_2 \in F_{k-1}$ s.t. $s_2 \neq s_1$ **do**
3: **if** $prefix(s_1) = prefix(s_2)$ **then**
4: **if** $isGenerator(s_1)$ **then**
5: $c \leftarrow concat(s_1, suffix(s_2))$
6: $C_k \leftarrow C_k \cup \{c\}$
7: **end if**
8: **if** $isGenerator(s_2)$ **then**
9: $c \leftarrow concat(s_2, suffix(s_1))$
10: $C_k \leftarrow C_k \cup \{c\}$
11: **end if**
12: **end if**
13: **end for**
14: **end for**
15: return C_k

Candidate Pruning: Candidate generation is followed by a pruning step. Pruned candidates are sequences identified as non-frequent based solely on the apriori property without calculating their support. Traditionally, [2,14], k-sequence candidates are only pruned if they have at least one $(k-1)$-subsequence that is not in F_{k-1}. However, our unique pruning strategy enables us to prune some of the k-sequence candidates for which this does not apply, due to the following observation: in the k^{th} iteration of the candidate generation step, it is possible that one k-sequence candidate will turn out to be a super-sequence of another k-sequence. This happens when events of the subsequence are contained in the corresponding events of the super-sequence. More formally, if we have the two k-sequences $s_1 = \langle e_1^1, e_2^1, ..., e_k^1\rangle$ and $s_2 = \langle e_1^2, e_2^2, ..., e_k^2\rangle$ then $s_1 \subseteq s_2$ if $e_1^1 \subseteq e_1^2 \wedge e_2^1 \subseteq e_2^2 \wedge ... \wedge e_k^1 \subseteq e_k^2$. If s_1 was found to be non-frequent, s_2 can be pruned, thereby avoiding the calculation of its support. For example, if the candidate $\langle aac\rangle$ is not frequent, we can prune $\langle(ab)ac\rangle$, which was generated in the same iteration.

Our pruning algorithm is decribed as follows. We iterate over all candidates $c \in C_k$ in an ascending radix order (this does not require a radix sort of C_k because its members are generated in this order - see section 5.2). For each c, we check whether all of its $(k-1)$-subsequences are in F_{k-1}. If not, then c is

not frequent and we add it to the set P_k which is a radix ordered list of pruned k-sequences that we maintain throughout the pruning step. Otherwise, we check whether a subsequence c_p of c exists in P_k. If so then again c is not frequent and we prune it. However, in this case, there is no need to add c to P_k because any candidate that would be pruned by c will also be pruned by c_p. Note that this check can be done efficiently since it only requires $O(log(|P_k|) \cdot k)$ comparisons of the events in the corresponding positions of c_p and c. Furthermore, if there are any k-subsequences of c in C_k that need to be pruned, the radix order of the process ensures that they have been already placed in P_k prior to the iteration in which c is checked. Finally, if c has not been pruned, we calculate its support. If c has passed the minSup threshold then it is frequent and we add it to F_k. Otherwise, it is not frequent, and we add it to P_k. Pseudocode for the candidate pruning is presented in Algorithm 3.

Support Calculation: In order to efficiently calculate the support of the k-sequence candidates we generate an occurrence index data structure for each candidate. This index is identical to the occurrence index of events, except that the list of eids represent candidates and not single events. A candidate is represented by the eid of the last event in it. The index for candidate $s_3 \in C_k$ is generated by intersecting the indices of $s_1, s_2 \in F_{k-1}$ from which s_3 is derived. Denoting the indices of s_1, s_2 and s_3 as inx_1, inx_2 and inx_3 respectively, the index intersection operation $inx_1 \odot inx_2 = inx_3$ is defined as follows: for each pair $sid_1 \in inx_1$ and $sid_2 \in inx_2$, we denote their associated eids list as $eidList(sid_1)$ and $eidList(sid_2)$, respectivly. For each $eid_1 \in eidList(sid_1)$ and $eid_2 \in eidList(sid_2)$, where $sid_1 = sid_2$ and $eid_1 < eid_2$, we add eid_2 to $eidList(sid_1)$ as new entry in inx_3. For example, consider the 1-sequences s_1=< d > and s_2=< (cd) > from the example database of Table 1. Their indices, inx_1 and inx_2, are described in Figure 1a. The index inx_3 for the 2-sequence s_3=< $d(cd)$ > is generated as follows: (i) for sid_1=1 and sid_2=1, we have eid_1=0 and eid_2=5 which will cause sid=1 to be added to inx_3 and eid=5 to be added to $eidList(1)$ in inx_3; (ii) for sid_1=3 and sid_2=3, we have eid_1=0 and eid_2=11 which will cause sid=3 to be added to inx_3 and eid=11 to be added to $eidList(3)$ in inx_3. The resulting inx_3 is shown in Figure 1b. Notice that the support of s_3 can be obtained by counting the number of elements in the sid list of inx_3. Thus, the use of the occurrence index enables us to avoid any database scans which would otherwise be required for support calculation.

Handling the MaxGap Constraint: Consider the example database in Table 1, with $minSup = 0.5$ and $maxGap = 5$. Now, let us look at the following three 2-sequences: $\langle ab \rangle$, $\langle ac \rangle$, $\langle cb \rangle$, all in C_2 and have a support of 2 (see section 5.3 for more details). If we apply the *maxGap* constraint, the sequence $\langle ab \rangle$ is no longer frequent and will not be added to F_2. This will prevent the algorithm from generating $\langle acb \rangle$, which is a frequent 3-sequence that does satisfy the *maxGap* constraint. To overcome this problem, during the support calculation, we mark frequent sequences that satisfy the *maxGap* constraint as *generators*. Sequences that do not satisfy the *maxGap* constraint, but whose support is

Algorithm 3. Candidate Pruning
Input F_{k-1}: the set of frequent $(k-1)$-sequences.
C_k: the set of k-sequence candidates.
minSup: minimum support for a frequent pattern.
maxGap: maximum time gap between consecutive events.
Output F_k: the set of frequent k-sequence.

procedure prune($F_{k-1}, C_k, minSup, maxGap$)
1: $P_k \leftarrow \phi$
2: **for all** candidates $c \in C_k$ **do**
3: $isGenerator(c) \leftarrow$ **true**;
4: **if** $\exists s \subseteq c \wedge s$ is a $(k-1)$-sequence $\wedge s \notin F_{k-1}$ **then**
5: $P_k.add(c)$
6: continue
7: **end if**
8: **if** $\exists c_p \in P_k \wedge c_p \subset c$ **then**
9: continue
10: **end if**
11: **if** $!(sup(c) \geq minSup)$ **then**
12: $P_k.add(c)$
13: **else**
14: $F_k.add(c)$
15: **if** !(c satisfies $maxGap$) **then**
16: $isGenerator(c) \leftarrow$ **false**;
17: **end if**
18: **end if**
19: **end for**
20: return F_k

higher than *minSup*, are marked as *non-generators* and we refrain from pruning them. In the following iteration we generate new candidates only from frequent sequences that were marked as generators. The procedure *isGenerator* in Algorithm 3 returns the mark of a sequence (i.e., whether it is a generator or not).

5.3 Example

Consider the sequence database D given in Table 1 with *minSup* set to 0.6 (i.e. 2 sequences), *maxGap* set to 5 and the set $\{a, b\}$ is a Singleton.

Event-wise phase. Find all frequent events in D. They are: $\langle(a)\rangle : 3$, $\langle(b)\rangle : 2$, $\langle(c)\rangle : 3$, $\langle(d)\rangle : 3$, $\langle(ac)\rangle : 2$, $\langle(ad)\rangle : 2$, $\langle(bd)\rangle : 2$, $\langle(cd)\rangle : 2$ and $\langle(acd)\rangle : 2$, where $\langle(event)\rangle : support$ represents the frequent event and its support. Each of the frequent events is marked as a generator, and together they form the set F_1.

Sequence-wise phase. This phase iterates over the the candidates list until no more candidates are generated. *Iteration 1, step 1: Candidate generation.* F_1 performs a self join, and 81 candidates are generated: $\langle aa\rangle$, $\langle ab\rangle$, $\langle ac\rangle$, ..., $\langle a(acd)\rangle$, $\langle ba\rangle$, $\langle bb\rangle$, ..., $\langle(acd)(acd)\rangle$. Together they form C_2. *Iteration 1, step 2: Candidate pruning.* Let us consider the candidate $\langle aa\rangle$. All its 1-subsequences

appear in F_1, so the F_{k-1} pruning passes. Next, P_2 still does not contain any sequences, so the P_k pruning passes as well. However, it does not pass the frequency test, since its support is 0, so $\langle aa \rangle$ is added to P_2. Now let us consider the candidate $\langle a(ac) \rangle$. It passes the F_{k-1} pruning, however, it does **not** pass the P_k pruning since a subsequence of it, $\langle aa \rangle$, appears in P_2. Finally, let us consider the candidate $\langle bc \rangle$. It passes both F_{k-1} and P_k pruning steps. Since it has a frequency of 2 it passes the frequency test, however, it does not satisfy the *maxGap* constraint, so it is marked as a non-generator, but added to F_2. All the sequences marked as generators in F_2 are: $\langle ac \rangle : 2$, $\langle cb \rangle : 2$, $\langle cd \rangle : 2$ and $\langle c(bd) \rangle : 2$. The other sequences in F_2 are marked as non-generators and are: $\langle ab \rangle : 2$, $\langle ad \rangle : 2$, $\langle a(bd) \rangle : 2$, $\langle dc \rangle : 2$, $\langle dd \rangle : 2$ and $\langle d(cd) \rangle : 2$.

Iteration 2: At the end of this iteration, only one candidate passes all the pruning steps: $\langle acb \rangle : 2$, and since no candidates can be generated from one sequences, the process stops.

6 Experimental Results

In order to evaluate the performance of CAMLS, we implemented SPADE, PrefixSpan (and its constrained version, Prefix-growth) and CAMLS in Java 1.6 using the Weka [5] platform. We compared the run-time of the algorithms by applying them on both synthetic and real data sets. We conducted several runs with and without including constraints. Since cSPADE does not incorporate all of the constraints we have defined, we have excluded it from the latter runs. All tests were conducted on an AMD Athlon 64 processor box with 1GB of RAM and a SATA HD running Windows XP. The synthetic datasets mimic real-world behavior of a Quartz-Tungsten-Halogen lamp. Such lamps are used in the semiconductors industry for finding defects in a chip manufacturing process. Each dataset is organized in a table of synthetically generated illumination intensity values emitted by a lamp. A row in the table represents a specific day and a column represents a specific wave-length. The table is divided into blocks of rows where each block represents a lamp's life cycle, from the day it is

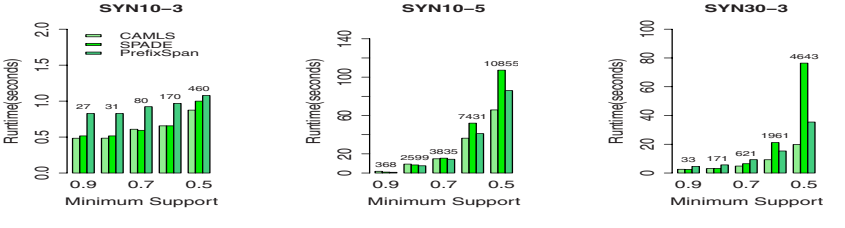


Fig. 2. Execution-time comparisons between CAMLS, SPADE and PrefixSpan on synthetic datasets for different values of minimum support, without constraints. The numbers appearing on top of each bar state the number of frequent patterns that exist for the corresponding minimum support.

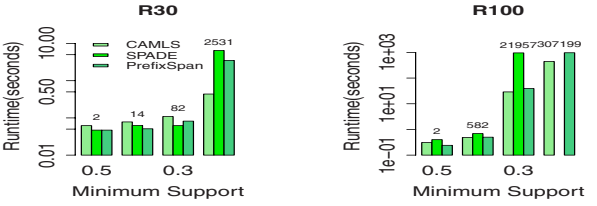


Fig. 3. Execution-time comparisons between CAMLS, SPADE and PrefixSpan on two real datasets for different values of minimum support, without constraints. The numbers appearing on top of each bar state the number of frequent patterns that exist for the corresponding minimum support.

first used to the last day it worked right before it burned out. To translate this data into events and sequences, the datasets were preprocessed as follows: first, the illumination intensity values were discretized into 50 bins using equal-width discretization [12]. Next, 5 items were generated: (i) the highest illumination intensity value out of all wave-lengths, (ii) the wave-length at which the highest illumination intensity value was received (iii) an indication whether or not the lamp burned out at the end of that day, (iv) the magnitude of the light intensity gradient between two consecutive measurements and (v) the direction of the light intensity gradient between two consecutive measurements. We then created two separate datasets. For each row in the original dataset, an event consisting of the first 3 items was formed for the first dataset and an event consisting of all 5 items was formed for the second one. Finally, in each dataset, a sequence was generated for every block of rows representing a lamp's life cycle from the events that correspond to these rows. We experimented with four such datasets each containing 1000 sequences and labeled SYNα-β where α stands for the sequence length and β stands for the event length. The real datasets, R30 and R100, were obtained from a repository of stock values [13]. The data consists of the values of 10 different stocks at the end of the business day, for a period of 30 or 100 days, respectively. The value of a stock for a given day corresponds to an event and the data for a given stock corresponds to a sequence, thus giving 10 sequences of either 30 (in R30) or 100 (in R100) events of length 1. As a preprocessing step, all numeric stock values were discretized into 50 bins using equal-frequency discretization [12].

Figure 2 compares CAMLS, SPADE and PrefixSpan on three synthetic datasets without using constraints. Each graph shows the change in execution-time as the minimum support descends from 0.9 to 0.5. The amount of frequent patterns found for each minimum support value is indicated by the number that appears above the respective bar. This comparison indicates that CAMLS has a slight advantage on datasets of short sequences with short events (SYN10-3). However, on datasets containing longer sequences (SYN30-3), the advantage of CAMLS becomes more pronounced as the amount of frequent patterns rises when decreasing the minimum support (around 5% faster than PrefixSpan and 35% faster than SPADE on the avarage). This is also true for datasets containing

longer events (SYN10-5) despite the lag in the proccess (the advantage of CAMLS is gained only after lowering the minimum support below 0.7). This lag results from the increased length of events which causes the check for the containment relation in the pruning strategy (line 8 of 3) to take longer. Similar results can be seen in Figure 3 which compares CAMLS, SPADE and PrefixSpan on the two real datasets. In R30, SPADE and PrefixSpan has a slight advantage over CAMLS when using high minimum support values. We believe that this can be attributed to the event-wise phase that slows CAMLS down, compared to the other algorithms, when there are few frequent patterns. On the other hand, as the minimum support decreases, and the number of frequent patterns increases, the performance of CAMLS becomes better by an order of magnitude. In the R100 dataset, where sequences are especially long, CAMLS clearly outperforms both algorithms for all minimum support values tested. In the extreme case of the lowest value of minimum support, the execution of SPADE did not even end in a reasonable amount of time. Figure 4 compares CAMLS and Prefix-growth on SYN30-3, SYM30-5 and R100 with the usage of the *maxGap* and Singletons constraints. On all three datasets, CAMLS outperforms Prefix-growth.

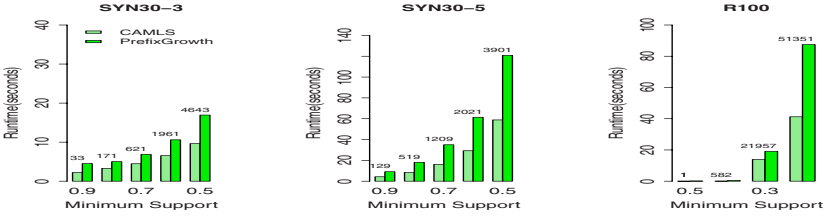


Fig. 4. Execution-time comparisons between CAMLS and Prefix-growth on SYN30-3 SYN30-5 and R100 with the *maxGap* and Singletons constraints for different values of minimum support. The numbers appearing on top of each bar state the number of frequent patterns that exist for the corresponding minimum support.

7 Discussion

In this paper we have presented CAMLS, a constraint-based algorithm for mining long sequences, that adopts the apriori approach. Many real-world domains require a substantial lowering of the minimum support in order to find any frequent patterns. This usually amounts to a large number of frequent patterns. Furthermore, some of these datasets may consist of many long sequences. Our motivation to develop CAMLS originated from realizing that well performing algorithms such as SPADE and PrefixSpan could not be applied on this class of domains. CAMLS consists of two phases reflecting a conceptual distinction between the treatment of temporal and non temporal data. Temporal aspects are only relevant during the sequence-wise phase while non temporal aspects are dealt with only in the event-wise phase. There are two primary advantages to this distinction. First, it allows us to apply a novel pruning strategy which accelerates

the mining process. The accumulative effect of this strategy becomes especially apparent in the presence of many long frequent sequences. Second, the incorporation of inter-event and intra-event constraints, each in its associated phase, is straightforward and the algorithm can be easily extended to include other inter-events and intra-events constraints. We have shown that the advantage of CAMLS over state of the art algorithms such as SPADE, PrefixSpan and Prefixgrowth, increases as the mined sequences get longer and the number of frequent patterns in them rises.

We are currently extending our results to include different domains and compare CAMLS to other algorithms. In future work, we plan to improve the CAMLS algorithm to produce only closed sequences and to make our pruning strategy even more efficient.

References

1. Agrawal, R., Srikant, R.: Fast Algorithms for Mining Association Rules. In: 20th Int. Conf. Very Large Data Bases, VLDB, pp. 487–499. Morgan Kaufmann, San Francisco (1994)
2. Zaki, M.J.: SPADE: An efficient algorithm for mining frequent sequences. Machine Learning Journal, special issue on Unsupervised Learning, 31–60 (2001)
3. Zaki, M.J.: Sequence mining in categorical domains: incorporating constraints. In: 9th Int. Conf. on Information and knowledge management, pp. 422–429. ACM, New York (2000)
4. Agrawal, R., Srikant, R.: Mining Sequential Patterns. In: 11th Int. Conf. Data Engineering, pp. 3–14. IEEE Computer Society, Los Alamitos (1995)
5. Witten, I.H., Frank, E.: Data mining: practical machine learning tools and techniques with Java implementations. J. SIGMOD Rec. 31(1), 76–77 (2002)
6. Han, J., Kamber, M.: Data Mining: Concepts and Techniques. Morgan Kaufmann, San Francisco (2006)
7. Pei, J., Han, J., Wang, W.: Constraint-based sequential pattern mining: the pattern-growth methods. J. Intell. Inf. Syst. 28(2), 133–160 (2007)
8. Pei, J., Han, J., Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Dayal, U., Hsu, M.: Mining sequential patterns by pattern-growth: The PrefixSpan approach. J. IEEE Transactions on Knowledge and Data Engineering 16 (2004)
9. Mannila, H., Toivonen, H., Verkamo, A.: Discovery of Frequent Episodes in Event Sequences. J. Data Min. Knowl. Discov. 1(3), 259–289 (1997)
10. Han, J., Pei, J., Yin, Y., Mao, R.: Mining Frequent Patterns without Candidate Generation A Frequent-Pattern Tree Approach. J. Data Min. Knowl. Discov. 8(1), 53–87 (2004)
11. Orlando, S., Perego, R., Silvestri, C.: A new algorithm for gap constrained sequence mining. In: The 2004 ACM symposium on Applied computing, pp. 540–547. ACM, New York (2004)
12. Pyle, D.: Data preparation for data mining. Morgan Kaufmann, San Francisco (1999)
13. Torgo, L.: Daily stock prices from January 1988 through October 1991, for ten aerospace companies, `http://www.liaad.up.pt/~ltorgo/Regression/DataSets.html`
14. Srikant, R., Agrawal, R.: Mining sequential patterns: Generalizations and performance improvements. In: 5th International Conference on Extending Database Technology. Springer, Heidelberg (1996)