ClaSP: An Efficient Algorithm for Mining Frequent Closed Sequences

Antonio Gomariz[1,⋆], Manuel Campos[2], Roque Marin[1], and Bart Goethals[3]

[1] Information and Communication Engineering Dept., University of Murcia, Spain
[2] Languages and Systems Dept., University of Murcia, Spain
[3] Mathematics and Computer Science Dept., University of Antwerp, Belgium

Abstract. In this paper, we propose a new algorithm, called ClaSP for mining frequent closed sequential patterns in temporal transaction data. Our algorithm uses several efficient search space pruning methods together with a vertical database layout. Experiments on both synthetic and real datasets show that ClaSP outperforms currently well known state of the art methods, such as CloSpan.

1 Introduction

Sequence Data Mining (SDM) is a well-extended field of research in Temporal Data Mining that consist of looking for a set of patterns frequently enough occurring across time among a large number of objects in a given input database. The threshold to decide if a pattern is meaningful is called *minimum support.* SDM has been widely studied [6,9,3,5,2], with broad applications, such as the discovery of motifs in DNA sequences, analysis of customer purchase sequences, web click streams, and so forth.

The task of discovering the set of all frequent sequences in large databases is challenging as the search space is extremely large. Different strategies have been proposed so far, among which *SPADE*, exploiting a vertical database format [9], FreeSpan and PrefixSpan, based on projected pattern growth [3,5] are the most popular ones. These strategies show good performances in databases containing short frequent sequences or when the support threshold is not very low. Unfortunately, when long sequences are mined, or when a very low support threshold is used, the performance of such algorithms decreases dramatically and the number of frequent patterns increases sharply, resulting in too many meaningless and redundant patterns. Even worse, sometimes it is impossible to complete the algorithm execution due to a memory overflow.

One of the most interesting proposals to solve both problems are so called *closed sequences* [8], based on the same notion for regular frequent closed itemsets, as introduced by Pasquier et al. [4]. A frequent sequence is said to be closed,

⋆ Corresponding author: `agomariz@um.es`. This work is partially financed by a PhD grant from the Seneca Foundation (Regional Agency for Science and Technology of the Region de Murcia), and by the Spanish Office for Science and Innovation through project TIN2009-14372-C03-01 and PlanE, and the European Union by means of the European Regional Development Fund (ERDF, FEDER).

if there no exists a supersequence with the same support in the database. The final collection of closed sequences provides a much more simplified output, still keeping all the information about the frequency of each of the sequences. Some algorithms have been developed to find the complete set of closed sequences, where most of them are based on the Pattern Growh strategy [8,7].

In this paper, we propose a new algorithm, called ClaSP (Closed Sequential Patterns algorithm) which exploits several efficient search space pruning methods. Depending on the properties of the database, we argue about the desirability of using the vertical database format as compared to pattern growth techniques. We also show the suitability of the vertical database format in obtaining the frequent closed sequence set, and how, under some database configurations, a standard vertical database format algorithm can already be faster than Pattern Growth algorithms for closed sequences, by only adding a simple post-processing step. Experiments on both synthetic and real datasets show that ClaSP generates the same complete closed sequences as CloSpan [8] but has much better performance figures.

The remaining of the paper is organized as follows. Section 2 introduces the preliminary concepts of frequent closed sequential pattern mining and the notation used in the paper. In Section 3, we present the most relevant related works. In Section 4, the pruning methods and ClaSP algorithm are presented. The performance study is presented in section 5 and, finally, we state our conclusions in section 6.

2 Problem Setting

Let $\mathcal{I}$ be a set of items. A set $X = \{e_1, e_2, \ldots, e_k\} \subseteq \mathcal{I}$ is called an itemset or k-itemsets if it contains k items. For simplicity, from now on we denote an itemset I as a concatenation of items between brackets. So, $I_1 = (ab)$ and $I_2 = (bc)$ are both two 2-itemsets. Also, without loss of generality, we assume the items in every itemset are represented in a lexicographic order.

A sequence s is a tuple $s = \langle I_1 I_2 \ldots I_n \rangle$ with $I_i \in \mathcal{I}$, and $\forall i : 1 \leq i \leq n$. We denote the size of a sequence $|s|$ as the number of itemsets in that sequence. We denote the length of a sequence ($l = \sum_{i=1}^{n} |I_i|$) as the number of items in it, and every sequence with k items is called a k-sequence. For instance, the sequence $\alpha = \langle (ab)(bc) \rangle$ is a 4-sequence with a size of 2 itemsets.

We say $\alpha = \langle I_{a_1} I_{a_2} \ldots I_{a_n} \rangle$ is a subsequence of another sequence $\beta = \langle I_{b_1} I_{b_2} \ldots I_{b_m} \rangle$ (or β is a supersequence of α), denoted as $\alpha \preceq \beta$, if there exist integers $1 \leq j_1 < j_2 < \ldots < j_n \leq m$ such that $I_{a_1} \subseteq I_{b_{j_1}}, I_{a_2} \subseteq I_{b_{j_2}}, \ldots, I_{a_n} \subseteq I_{b_{j_n}}$. For instance, $\langle (b)(c) \rangle$ is a subsequence of $\langle (ab)(bc) \rangle$, since $(b) \subseteq (ab)$ and $(c) \subseteq (bc)$ and the order in the itemsets is preserved. Furthermore, the sequence $\langle (b)(c) \rangle$ is not a subsequence of $\langle (abc) \rangle$.

In the rest of the work, we use the terms pattern and sequence interchangeably.

An input sequence is is a tuple $is = \langle id, s \rangle$ with $id \in \mathbb{N}$ and s is a sequence. We call id the identifier of the input sequence. We say that an input sequence is contains another sequence α, if $\alpha \preceq s$.

Table 1. A Sample Sequence Database

Sequence Id.	Sequence
1	$\langle(a)(ab)(bc)\rangle$
2	$\langle(a)(abc)\rangle$
3	$\langle(d)(a)(ab)(bc)\rangle$
4	$\langle(d)(ad)\rangle$

A sequence database D is collection of input sequences $D = \langle s_1 s_2 \ldots s_n\rangle$, incrementally ordered by the identifier of the contained sequences. In table 1 we show a sample input database D with four input sequences.

Definition 1. The *support* (or *frequency*) of a sequence, denoted as $\sigma(\alpha, D)$, is the total number of input sequences in the input database D that contain α. A pattern or sequence is called *frequent* if it occurs at least a given user specified threshold *min_sup*, called the minimum support. $\mathcal{FS}$ is the whole collection of frequent sequences. The problem of frequent sequence mining is now to find $\mathcal{FS}$ in a given input database, for a given minimum support threshold.

Given a sequence $\alpha = \langle I_1 I_2 \ldots I_n\rangle$ and an item e_i, we define the *s-extension* α' as the super-sequence of α, extending it with a new itemset containing a single item e_i, $\alpha' = \langle I_1 I_2 \ldots I_n I_{n+1}\rangle$, $I_{n+1} = (e_i)$. We define the *i-extension* of α if the last itemset I'_n of $\alpha' = \langle I_1 I_2 \ldots I'_n\rangle$ satisfies ($I'_n = I_n \cup e_i$). That is, the item e_i is added to I_n. For instance, given the sequence $\alpha = \langle(a)(b)\rangle$ and an item $c \in \mathcal{I}$, the sequence $\beta = \langle(a)(b)(c)\rangle$ is an s-extension and $\gamma = \langle(a)(bc)\rangle$ is an i-extension.

Given two sequences β and γ such that both are s-extensions (or i-extensions) of a common prefix α, with items e_i and e_j respectively, we say β precedes γ, $\beta < \gamma$, if $e_i <_{lex} e_j$ in a lexicographic order. If, on the contrary, one of them is an s-extension and the other one is i-extension, the s-extension always precedes the i-extension.

Definition 2. If a frequent sequence α does not have another supersequence with the same support, we say that α is a *closed* sequence. Otherwise, if a frequent sequence β has a super-sequence γ with exactly the same support, we say that β is a non-closed sequence and γ *absorbs* β. The whole set of frequent closed sequences is denoted by $\mathcal{FCS}$. More formally, $\alpha \in \mathcal{FCS}$ if $\forall \beta \in \mathcal{FS}, \alpha \preceq \beta, \sigma(\alpha, D) \neq \sigma(\beta, D)$. The problem of closed sequence mining is now to find $\mathcal{FCS}$ in a given input database, for a given minimum support threshold.

Clearly, the collection of frequent closed sequences is smaller than the collection of all frequent sequences.

Example 1. *In our sample database, shown in table 1, for a support min_sup = 2, we find $|\mathcal{FCS}| = 5$ frequent closed sequences, $\mathcal{FCS} = \{\langle(a)\rangle, \langle(d)(a)\rangle, \langle(a)(ab)\rangle, \langle(a)(bc)\rangle, \langle(a)(ab)(bc)\rangle\}$, while the corresponding $\mathcal{FS}$ has 27 frequent sequences.*

3 Related works

Looking for frequent sequences in sequence databases was first proposed by Agrawal and Srikant [1,6]. Their algorithms (apriori-based) consist of executing a continuous loop of a candidate generation phase followed by a support checking phase. Two main drawbacks appear in those algorithms: 1) they need to do several scans of the database to check the support of the candidates; and 2) a breath-first search is needed for the candidate generation, leading to high memory consumption.

Later, two other strategies were proposed: 1) depth-first search based on a vertical database format [9] and 2) projected pattern growth [3,5]. The vertical database format strategy was created by Zaki in the Spade algorithm [9] which is capable of obtaining the frequent sequences without making several scans of the input database. His algorithm allows the decomposing of the original search tree in independent problems that can be solved in parallel in a depth-first search (DFS), thus enabling the processing of big databases.

The Pattern growth strategy was introduced by Han et al. [3] and it consists in algorithms that obtain the whole frequent sequence set by mean of techniques based on the so called projected pattern growth. The most representative algorithm in this strategy is PrefixSpan [5]. PrefixSpan defines a projected database as the set of suffixes with respect to a given prefix sequence. After projecting by a sequence, new frequent items are identified. This process is applied in a recursive manner by means of DFS, identifying the new frequent sequences as the concatenation of the prefix sequence with the frequent items that are found.

Prefixspan shows good performance and scales well in memory, especially with sparse databases or when databases mainly consist of small itemsets. However, when we deal with large dense databases that have large itemsets, the performance of Prefixpan is worse than that of Spade. In order to show this issue, we have conducted several tests. In a sequential database, several important properties have an influence on the algorithms execution, some of which are shown in Table 2.

Table 2. Parameters for IBM Quest Data Generator

Abbr.	Meaning
D	Number of sequences (in 000s)
C	Average itemset in a sequence
T	Average items in a itemset
N	Number of different items (in 000s)
S	Average itemsets in maximal sequences
I	Average items in maximal sequences

We define the database density as the quotient $\delta = \frac{T}{N}$. We have used the well-known data generator provided by IBM to run Spade and PrefixSpan under different configurations. In Figures 1 and 2 we can observe the behaviour of both Spade and Prefixspan when we vary the density and the number of itemsets.

In figure 1 we show the running time of the algorithms with a different number of items (100, 500 and 2500 items) with a constant $T = 20$ value. Since for a database, the density grows either if the numerator increases or the denominator decreases, the figures have been obtained just varying the denominator. Besides, figure 2 depicts the behaviour of the algorithms when the number of itemsets is changed between values of $C \in \{10, 40, 80\}$ while we keep the density constant ($\delta = \frac{20}{2500}$). The higher δ the more dense the database. We can see that PrefixSpan shows good results when both density and the number of itemsets are low, but when a database is denser and parameter C grows, we notice how Spade outperforms Prefixspan.

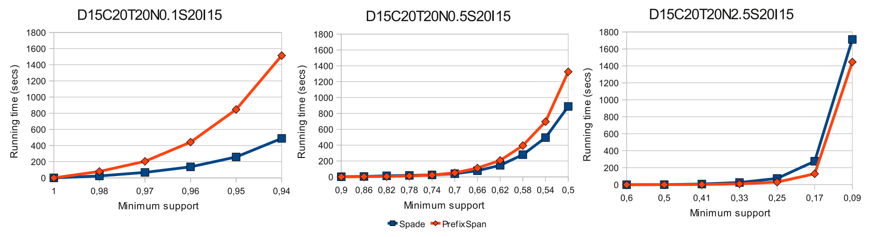

Fig. 1. Behaviour of Spade and PrefixSpan when density changes (in the number of items)

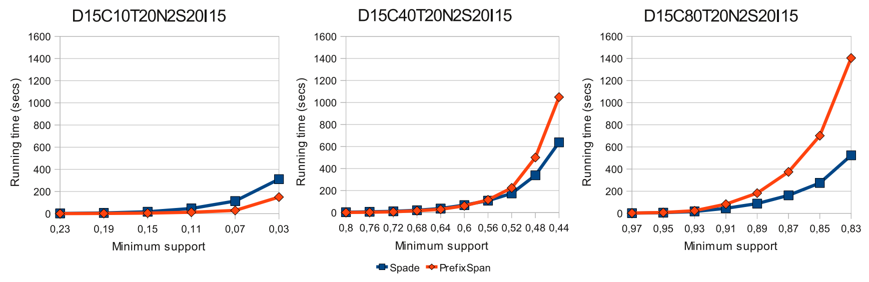

Fig. 2. Behaviour of Spade and PrefixSpan when the number of itemsets changes

For mining closed sequences, there exist two approaches: 1) run any algorithm for mining all frequent sequences and execute a post-processing step to filter out the set of closed sequences, or 2) obtain the set of closed sequences by gradually discarding the non-closed ones. Some algorithms have been developed to find the complete set of closed sequences. The most important algorithms developed so far, are CloSpan [8] and Bide [7], both derived from Prefixspan. While CloSpan uses a prefix tree to store the sequences and uses two methods to prune non-frequent sequences, Bide executes some checking steps in the original database that allows it to avoid maintaining the sequence tree in memory. However, to the best of our knowledge, there exist no algorithms for closed sequence mining based on the vertical database format as is presented here.

4 ClaSP: Algorithm and Implementation

In this section, we formulate and explain every step of our ClaSP algorithm. ClaSP has two main phases: The first one generates a subset of $\mathcal{FS}$ (and superset of $\mathcal{FCS}$) called Frequent Closed Candidates ($\mathcal{FCC}$), that is kept in main memory; and the second step executes a post-pruning phase to eliminate from $\mathcal{FCC}$ all non-closed sequences to finally obtain exactly $\mathcal{FCS}$.

Algorithm 1. ClaSP

```
1: $\mathcal{F}_1$ = {frequent 1-sequences}
2: $\mathcal{FCC} = \emptyset$, $\mathcal{FCS} = \emptyset$
3: for all  i $\in \mathcal{F}_1$  do
4:     $\mathcal{F}_{ie}$ = {frequent 1-sequences greater than i}
5:     $\mathcal{FCC}_i$=DFS-PRUNING(i,$\mathcal{F}_1$,$\mathcal{F}_{ie}$)
6:     $\mathcal{FCC} = \mathcal{FCC} \cup \mathcal{FCC}_i$
7: end for
8: $\mathcal{FCS}$ = N-ClosedStep($\mathcal{FCC}$)
Ensure: The final closed frequent pattern set $\mathcal{FCS}$
```

Algorithm 1, *ClaSP*, shows the pseudocode corresponding to the two main steps. It first finds every frequent 1-sequence, and after that, for all of frequent 1-sequences, the method *DFS-Pruning* is called recursively to explore the corresponding subtree (by doing a depth-first search). $\mathcal{FCC}$ is obtained when this process is done for all of the frequent 1-sequences and, finally, the algorithm ends removing the non-closed sequences that appear in $\mathcal{FCC}$.

Algorithm 2, *DFS-Pruning*, executes recursively both the candidate generation (by means of i-extensions and s-extensions) and the support checking, returning a part of $\mathcal{FCC}$ relative to the pattern p taken as parameter. The method takes as parameters two sets with the candidate items to do s-extensions and i-extensions respectively ($\mathcal{S}_n$ and $\mathcal{I}_n$ sets). The algorithm first checks if the current pattern p can be discarded, by using the method *checkAvoidable* (this algorithm is explained later in algorithm 5). Lines 4-9 perform all the s-extensions for the pattern p and keep in $\mathcal{S}_{temp}$ the items which make frequent extensions. In line 10, the method *ExpSiblings* (algorithm 4) is called, and there, *DFS-Pruning* is executed for each new frequent s-extensions. Lines 11-16 and 17 perform the same steps, with i-extensions. Finally, in line 19, the complete frequent patterns set (with a prefix p) is returned.

To store the patterns in memory, we use a lexicographic sequence tree. The elements in the tree are sorted by a lexicographic order according to extension comparisons (see section 2). In figure 3 we show the associated sequence tree for $\mathcal{FS}$ in our example and we denote an s-extension with a line, and an i-extension with a dotted line. This tree is traversed by algorithms 1, 2 and 4, using a depth-first traversal.

There are two main different changes added in ClaSP with respect to SPADE: (1) the step to check if the subtree of a pattern can be skipped (line 3 of algorithm 2), and (2) the step where the remaining non-closed patterns are eliminated (line 6 of algorithm 1).

Algorithm 2. DFS-Pruning(p, $\mathcal{S}_n$, $\mathcal{I}_n$)

```
Require: Current frequent pattern $p = (s_1, s_2, \ldots, s_n)$, set of items for s-extension $\mathcal{S}_n$, set of items for i-extension $\mathcal{I}_n$
1: $\mathcal{S}_{temp} = \emptyset$, $\mathcal{I}_{temp} = \emptyset$
2: $\mathcal{F}_i = \emptyset$, $\mathcal{P}_s = \emptyset$, $\mathcal{P}_i = \emptyset$
3: if (not checkAvoidable( $p$, $\mathcal{I}(\mathcal{D}_p)$ )) then
4:     for all i $\in \mathcal{S}_n$ do
5:         if ($p' = (s_1, s_2, \ldots, s_n, \{i\})$ is frequent) then
6:             $\mathcal{S}_{temp} = \mathcal{S}_{temp} \cup \{i\}$
7:             $\mathcal{P}_s = \mathcal{P}_s \cup \{p'\}$
8:         end if
9:     end for
10:    $\mathcal{F}_i = \mathcal{F}_i \cup \mathcal{P}_s \cup$ ExpSiblings($\mathcal{P}_s$,$\mathcal{S}_{temp}$,$\mathcal{S}_{temp}$)
11:    for all i $\in \mathcal{I}_n$ do
12:        if ($p' = (s_1, s_2, \ldots, s_n \cup \{i\})$ is frequent) then
13:            $\mathcal{I}_{temp} = \mathcal{I}_{temp} \cup \{i\}$
14:            $\mathcal{P}_i = \mathcal{P}_i \cup \{p'\}$
15:        end if
16:    end for
17:    $\mathcal{F}_i = \mathcal{F}_i \cup \mathcal{P}_i \cup$ ExpSiblings($\mathcal{P}_s$,$\mathcal{S}_{temp}$,$\mathcal{I}_{temp}$)
18: end if
19: return $\mathcal{F}_i$
Ensure: Frequent pattern set $\mathcal{F}_i$ of this node and its siblings
```

Algorithm 3. N-ClosedStep($\mathcal{FCC}$)

```
Require: A frequent closed candidates set $\mathcal{FCC}$
1: $\mathcal{FCS} = \emptyset$
2: A hash table $H$ is created
3: for all $p \in \mathcal{FCC}$ do
4:     Add a new entry $\langle \mathcal{T}(\mathcal{D}_p), p \rangle$ in $H$
5: end for
6: for all entry $e \in H$ do
7:     for all $p_i \in e$ do
8:         for all $p_j \in e$, $j > i$ do
9:             if ($p_i.support() = p_j.support()$) then
10:                if ($p_i \preceq p_j$) then
11:                    Remove $p_i$ from $e$
12:                else
13:                    if ($p_j \preceq p_i$) then
14:                        Remove $p_j$ from $e$
15:                    end if
16:                end if
17:            end if
18:        end for
19:    end for
20:    $\mathcal{FC}_e$= all patterns $p \in e$
21:    $\mathcal{FCS} = \mathcal{FCS} \cup \mathcal{FC}_e$
22: end for
23: return $\mathcal{FCS}$
Ensure: The final closed frequent pattern set $\mathcal{FCS}$
```

Algorithm 4. ExpSiblings($\mathcal{P}$, $\mathcal{S}_s$, $\mathcal{S}_i$)

```
Require: The pattern set $\mathcal{P}$ which contains all the patterns whose children are going to be explore, the set of valid items $\mathcal{S}_s$ which generate the $\mathcal{P}$ set by means of s-extensions, the set of valid items $\mathcal{S}_i$ which generate the $\mathcal{P}$ set by means of i-extensions
1: $\mathcal{F}_s = \emptyset$
2: for all p $\in \mathcal{P}$ do
3:     $\mathcal{I}$ = Elements in $\mathcal{S}_i$ greater than the last item $e_i$ in $p$
4:     $\mathcal{F}_s = \mathcal{F}_s \cup$ DFS-Pruning($p$,$\mathcal{S}_s$,$\mathcal{I}$)
5: end for
6: return $\mathcal{F}_s$
Ensure: Frequent pattern set $\mathcal{F}_s$ for all of the patterns' siblings
```

Algorithm 5. CheckAvoidable(p, k)

```
Require: a frequent pattern $p$, its hash-key $k$, hash table $H$
1: $\mathcal{M}_k$ = Entries in the hash table with key $k$
2: if ($\mathcal{M}_k = \emptyset$) then
3:     Insert a new entry $\langle k$,$p \rangle$ in $H$
4: else
5:     for all pair $m \in \mathcal{M}_k$ do
6:         $p' = m.value()$
7:         if ($p$.support()=$p'.support()$) then
8:             //Backward sub-pattern
9:             if ($p \preceq p'$) then
10:                $p$ has the same descendants as $p'$, so $p$ points to $p'$ descendants
11:                return true
12:            else
13:                //Backward super-pattern
14:                if ($p' \preceq p$) then
15:                    $p$ has the same descendants as $p'$, so $p$ points to $p'$ descendants
16:                    Remove the current entry $\langle k, p' \rangle$ from $H$
17:                end if
18:            end if
19:        end if
20:    end for
21:    //Backward super-pattern executed?
22:    if (Pruning method 2 has been accomplished) then
23:        Add a new entry $\langle k, p \rangle$ in $H$
24:        return true
25:    end if
26: end if
27: //k does not exist in the hash table or it does exist but the present patterns are not related with p
28: Add a new entry $\langle k, p \rangle$ in $H$
29: return false
Ensure: It answers if the generation of $p$ can be avoided
```

To prune the space search, ClaSP used the method *CheckAvoidable* that is inspired on the pruning methods used in CloSpan. This method tries to find those patterns $p = \langle \alpha \, e_j \rangle$ and $p' = \langle \alpha \, e_i \, e_j \rangle$, such that, all of the appearances of p are in those of p', i.e., if every time we find a sequence α followed by an item e_j, there exists an item e_i between them, then we can safely avoid the exploration of the subtree associated to the pattern p. In order to find this kind of patterns, we define two numbers: 1) $l(s, p)$, is the size of all the suffixes with respect to p in sequence s, and 2) $\mathcal{I}(\mathcal{D}_p) = \sum_{i=1}^{n} l(s_i, p)$, the total number of remaining items with respect to p for the database $\mathcal{D}$, i.e. the addition of all of $l(s, p)$ for every sequence in the database. Using $\mathcal{I}(\mathcal{D})$ and the subsequence checking operation, in algorithm 5, ClaSP checks the equivalence between the $\mathcal{I}(\mathcal{D})$ values for two patterns: Given two sequences, s and s', such that $s \preceq s'$, if $\mathcal{I}(\mathcal{D}_s) = \mathcal{I}(\mathcal{D}_{s'})$, we can deduce that the support for all of their descendants is just the same.

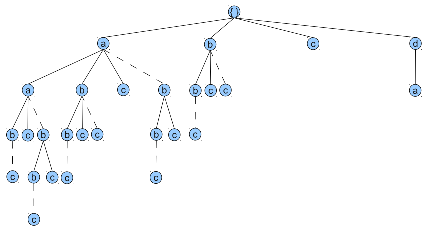

Fig. 3. Whole lexicographic sequence tree for our thorough example

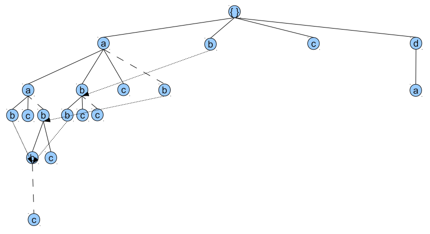

Fig. 4. Whole lexicographic sequence tree after processing ClaSP algorithm

In algorithm 5, the pruning phase is implemented by two methods: 1) Backward sub-pattern checking and 2) Backward super-pattern checking. The first one (lines 8-10) occurs when we find a pattern which is a subsequence of a pattern previously found with the same $\mathcal{I}(\mathcal{D})$ value. In that case, we can avoid exploring this new branch in the tree for this new pattern. The second method (lines 12-16 and 20-24) is the opposite situation and it occurs when we find a pattern that is a super-sequence of another pattern previously found with the same $\mathcal{I}(\mathcal{D})$ value. In this case we can transplant the descendants of the previous pattern to the node of this new pattern.

In figure 4 we show the ClaSP search tree w.r.t. our example without all pruned nodes. In our implementation, to store the relevant branches, we define a hash function with $\mathcal{I}(\mathcal{D})$ value as key and the pattern (i.e. the node in the tree for that pattern) as value ($\langle \mathcal{I}(\mathcal{D}_p), p \rangle$). We use a global hash table and, when we find a sequence p, if the backward sub-pattern condition is accomplished, we do not put the pair $\langle \mathcal{I}(\mathcal{D}_p), p \rangle$, whereas, if the backward super-pattern condition is true, we replace all the previous pairs $\langle \mathcal{I}(\mathcal{D}_{p'}), p' \rangle$ (s.t. $p' \preceq p$) with the new one $\langle \mathcal{I}(\mathcal{D}_p), p \rangle$. If instead we do not find any pattern with the same $\mathcal{I}(\mathcal{D})$ value of the pattern p, or those patterns with the same value are not related with p by means of the subsequence operation, we put the pair $\langle \mathcal{I}(\mathcal{D}_p), p \rangle$ to the global hash table (line 28). Note that when one of the two pruning conditions is true, we also need to check if the support for s and s' is the same since two $\mathcal{I}(\mathcal{D}_p)$ and $\mathcal{I}(\mathcal{D}_{p'})$ values can be equal but they do not necessarily have the same support.

We also need to consider all of the $\mathcal{I}(\mathcal{D})_s$ for every appearance in a sequence. For instance, in our example shown in table 1, regarding the three first sequences, if we consider just the first $\mathcal{I}(\mathcal{D}_s)$ for the first appearance, if we have the pattern $\langle(a)(b)\rangle$ in our example, we deduce that $\mathcal{I}(\mathcal{D}_{\langle(a)(ab)\rangle}) = \mathcal{I}(\mathcal{D}_{\langle(a)(b)\rangle})$ (both with value $\mathcal{I}(\mathcal{D}) = 5$), so we can avoid generating the descendants of $\langle(a)(b)\rangle$ because the are the same as in $\langle(a)(ab)\rangle$. However, we can check that $\langle(a)(bc)\rangle$ is frequent (with support 3), whereas $\langle(a)(abc)\rangle$ is not (support 1). This forces us to count in $\mathcal{I}(\mathcal{D}_s)$, all the number of items after every appearance.

Finally, regarding the non-closed pattern elimination (algorithm 3), the process consists of using a hash function with the support of a pattern as key and the pattern itself as value. If two patterns have the same support we check if one contains the other, and if this condition is satisfied, we remove the shorter pattern. Since the support value as key provoke a high number of collisions in the hash table (implemented with closed addressing), we use a $\mathcal{T}(\mathcal{D}_p) = \sum_{i=1}^{n} id(s_i)$ value, defined as the sum of all sequence ids where a pattern p appears. However, as the equivalence of $\mathcal{T}(\mathcal{D}_p)$ does not imply the equivalence of support, after checking that two patterns have the same $\mathcal{T}(\mathcal{D}_p)$ value, those patterns have to have the same support to remove one of them.

5 Performance Study

We exhaustively experimented on both synthetic and real world datasets. To generate the synthetic data, we have used the IBM data generator mentioned above (see section 3). In all our experiments we compare the performance of three algorithms: CloSpan, ClaSP and Spade. For the last algorithm we add the same non-closed candidate elimination phase which is used in ClaSP to obtain $\mathcal{FCS}$.

All experiments are done on a 4-cores of 2.4GHZ Intel Xeon, running Linux Ubuntu 10.04 Server edition. All the three algorithms are implemented in Java 6 SE with a Java Virtual Machine of 16GB of main memory.

Figure 5 shows the number of patterns and performance for the dataset D5C10T5N5S6I4 (-rept 1 -seq.npats 2000 -lit.npats 5000). Figure 5(a) shows the number of frequent patterns, the patterns processed by ClaSP, and the number of closed patterns. We can see how there is approximately an order of difference between these numbers, i.e. for every 100 frequent patterns we process around 10 patterns by ClaSP, and approximately only 1 of these 10 patterns is closed. Figure 5(b) shows the running time. ClaSP clearly outperforms both Spade and Clospan. For very low support (below 0.013), we have problems with the execution of Spade due to the space in memory taken for the algorithm.

Figure 6 shows a dataset with larger parameters of C, T and a lower N. This database is denser than the database above and in figure 6(a) we can observe that the difference between frequent patterns and processed patterns is not so big. Therefore, the pruning method checkAvoidable is not so effective and ClaSP

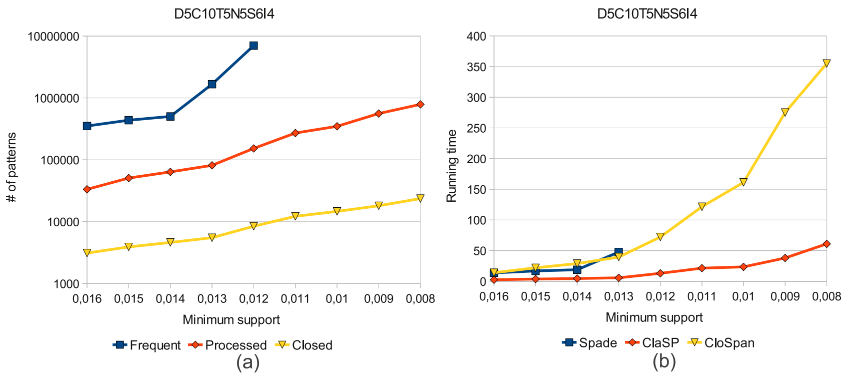

Fig. 5. Varying Support for Dataset D5C10T5N5S6I4(-seq.npats 2000 -lit.npats 5000)

and CloSpan are closer to the normal behavior of Spade and PrefixSpan, as is shown in figure 6(b). The results show that both ClaSP and Spade are much faster than Clospan.

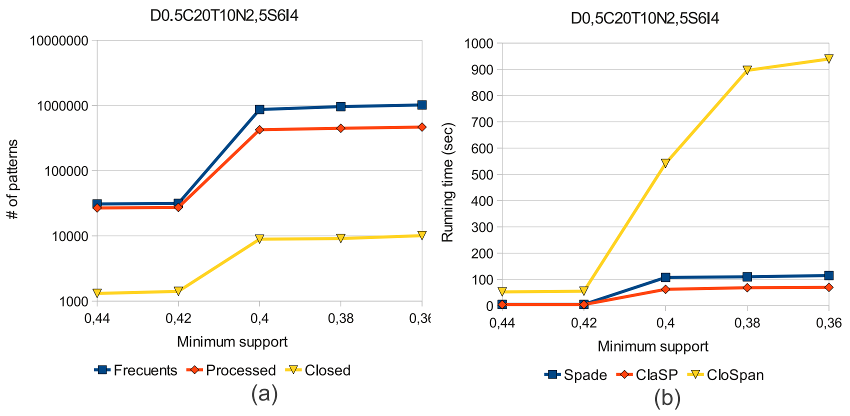

Fig. 6. Varying Support for Dataset D0.5C20T10N2.5S6I4(-seq.npats 2000 -lit.npats 5000)

Finally we test our algorithm with the gazelle dataset. This dataset comes from click-stream data from gazelle.com, which no longer exists. The dataset was once used in KDDCup-2000 competition and, basically, it includes a set of page views (each page contains a specific product information) in a legwear and legcare website. Each session contains page views done by a customer over a short period. Product pages viewed in one session are considered as an itemset, and different sessions for one user is considered as a sequence. The database contains 1423 different products and assortments which are viewed by 29369 different users. There are 29369 sequences, 35722 sessions (itemsets), and 87546 page views (items). The average number of sessions in a sequence is around 1. The average number of pageviews in a session is 2. The largest session contains 342 views, the longest sequence has 140 sessions, and the largest sequence contains 651 page views. Figure 7 shows the runtime with several support (from 0.03% to 0.015%). We compare the runtime behavior for both ClaSP and CloSpan and we can see how ClaSP outperforms CloSpan.

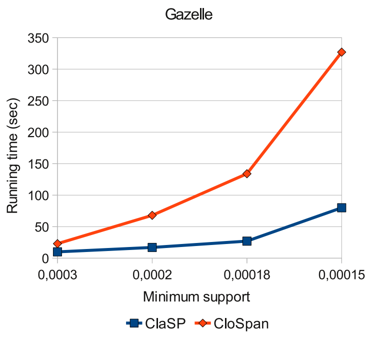

Fig. 7. Varying Support for Dataset Gazelle click stream

All the experiments show that ClaSP outperforms CloSpan, even if databases are sparse. This is because of, in very sparse databases, a high number of patterns are found only with extremely low support. Therefore, the lower the support is, the more patterns are found and the more items are chosen to create patterns. In this point, CloSpan, as PrefixSpan, is penalized since the algorithm has to projects several times the same item in the same sequence, having a worse time with respect to ClaSP. Besides, in those algorithms where Spade is better than Prefixspan, ClaSP is also faster than CloSpan.

6 Conclusions

In this paper, we study the principles for mining closed frequent sequential patterns and we compare the two main existing strategies to find frequent patterns. We show the benefits of using the vertical database format strategy against pattern growth algorithms, especially when facing dense datasets. Then, we introduced a new algorithm called ClaSP to mine frequent closed sequences. This algorithm is inspired on the Spade algorithm using a vertical database format strategy and uses a heuristic to prune non-closed sequences inspired by the CloSpan algorithm. To the best of our knowledge, this is the first work based on the vertical database strategy to solve the closed sequential pattern mining problem. In all our tests ClaSP outperforms CloSpan, and even, under certain datasets configuration, Spade also outperforms CloSpan when the non-closed elimination phase is executed after it.

References

1. Agrawal, R., Srikant, R.: Mining sequential patterns. In: Proceedings of the Eleventh International Conference on Data Engineering, pp. 3–14. IEEE (1995)
2. Ayres, J., Flannick, J., Gehrke, J., Yiu, T.: Sequential pattern mining using a bitmap representation. In: Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 429–435. ACM (2002)
3. Han, J., Pei, J., Mortazavi-Asl, B., Chen, Q.: FreeSpan: frequent pattern-projected sequential pattern mining. In: Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 355–359 (2000)
4. Pasquier, N., Bastide, Y., Taouil, R., Lakhal, L.: Discovering Frequent Closed Itemsets for Association Rules. In: Beeri, C., Bruneman, P. (eds.) ICDT 1999. LNCS, vol. 1540, pp. 398–416. Springer, Heidelberg (1998)
5. 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. IEEE Transactions on Knowledge and Data Engineering 16(11), 1424–1440 (2004)
6. Srikant, R., Agrawal, R.: Mining Sequential Patterns: Generalizations and Performance Improvements. In: Apers, P.M.G., Bouzeghoub, M., Gardarin, G. (eds.) EDBT 1996. LNCS, vol. 1057, pp. 3–17. Springer, Heidelberg (1996)
7. Wang, J., Han, J., Li, C.: Frequent closed sequence mining without candidate maintenance. IEEE Transactions on Knowledge and Data Engineering 19(8), 1042–1056 (2007)
8. Yan, X., Han, J., Afshar, R.: CloSpan: Mining closed sequential patterns in large datasets. In: Proceedings of SIAM International Conference on Data Mining, pp. 166–177 (2003)
9. Zaki, M.J.: SPADE: An efficient algorithm for mining frequent sequences. Machine Learning 42(1), 31–60 (2001)